THE DEVELOPMENT OF A NODE FOR A HARDWARE RECONFIGURABLE PARALLEL PROCESSOR

prepared by

Carl Frans van Schaik

A dissertation submitted to the Department of Electrical Engineering, University of Cape Town

in partial fulfilment of the requirements, for the degree of Master of Science in Engineering

Cape Town, January 2002

Last edit: Wed, 6 Mar 2002 8:59

Declaration

I, Carl Frans van Schaik declare that all information contained within this dissertation report is original and my own work to the best of my knowledge, except were indicated in the text. This dissertation report has not been submitted, in whole or part to any organisation other than those directly related to the University of Cape Town.

signed

_____ _____

Carl Frans van Schaik Date

Acknowledgements

I would like to thank all the people for assisting in the completion of this project as well as:

My supervisor Professor M.R. Inggs for his time, dedication and supporting of this project.

Alan Langman for being my 'unofficial' supervisor and who provided much input and focus for the project not to mention a lot of patience.

Abstract

This dissertation concerns the design and implementation of a node for a hardware reconfigurable parallel processor. The hardware that was developed allows for the further development of a parallel processor with configurable hardware acceleration. Each node in the system has a standard microprocessor and reconfigurable logic device and has high speed communications channels for inter-node communication.

The design of the node provided high-speed serial communications channels allowing the implementation of various network topographies. The node also provided a PCI master interface to provide an external interface and communicate with local nodes on the bus. A high speed RISC processor provided communication and system control functions and the reconfigurable logic device provided communication interfaces and data processing functions.

The node was designed and implemented as a PCI card that interfaced a standard PCI bus. VHDL designs for logic devices that provided system support were developed, VHDL designs for the reconfigurable logic FPGA and software including drivers and system software were written for the node. The 64-bit version Linux operating system was then ported to the processor providing a UNIX environment for the system.

The node functioned as specified and parallel and hardware accelerated processing was demonstrated. The hardware acceleration was shown to provide substantial performance benefits for the system.

Contents

List of Figures

LIST OF FIGURES

List of Tables

Glossary

ASIC Application Specific Integrated Circuit: An integrated circuit that has been specially designed for a custom application, typically proprietary.

CPLD Complex Programmable Logic Device: A programmable logic device similar to, but providing more features, such as flip-flops and feedback, than PALs and GALs.

DCT Discrete Cosine Transform: A mathematical algorithm to calculate the Cosine function components of a signal represented by a discrete set of points.

FFT Fast Fourier Transform: A mathematical algorithm for converting time domain data into frequency domain data.

FPGA Field Programmable Gate Array: A device providing a large array of configurable logic building blocks and configurable routing interconnects for implementing logic designs.

PVM Parallel Virtual Machine: A software library and application for running parallel processing programs on a heterogeneous network of machines.

Chapter 1

Introduction

The goal of this MSc project was the design, implementation and testing of a node for a run-time hardware reconfigurable parallel computing processor. Three major tasks were undertaken in the completion of this project: Analysing a selection of successful parallel computing architectures and evaluating their characteristics. Specifying, designing and implementing the node hardware according to an architecture derived from the first task with the added advantage of configurable logic. And lastly, implementing a software environment to provide a base for further parallel processing research and demonstrating basic proof-of-concept examples.

Configurable logic allows the logical circuitry of a specialised silicon chip, in particular FPGAs, to be configured and changed without modifying the physical devices in the system. FPGA devices allow for highly specialised digital designs to be implemented in general-purpose silicon without the cost of developing custom silicon. Reconfigurable logic allows the logic design configured in the device to be changed at any time, especially while the device and system are in operation.

Parallel processing is the use of multiple processing units connected together to handle an intensive computational task. Parallel processing is most commonly employed to reduce the processing time for certain very large or complicated processing tasks. This works because certain processing tasks can be achieved by allowing each processing node to work on a subset of the entire problem. Some example applications in which parallel processing is used are: Image processing, finite element simulations and computer generated animation.

In many processing tasks, a single or small number of algorithmic functions are used extensively on a large amount of data. In certain cases, the algorithm used can be implemented directly in a digital logic design in hardware. When this is possible, the hardware implementations are usually orders of magnitude faster than the same algorithm running on a digital microprocessor. The possibility exists to implement these hardware implementations in reconfigurable logic devices rather than custom silicon. The use of these devices with their ability to implement arbitrary configurations allows for a more general-purpose hardware accelerated processing unit to be designed. Further, by using these hardware accelerated processing functions in com-

bination with standard microprocessors, a general purpose computing platform can be designed. Extending this idea with the principles of parallel processing, a hardware accelerated parallel processor can be designed which can greatly improve on the processing speed of standard parallel processors. This is especially relevant for tasks that employ hardware implementable processing algorithms.

The Radar Remote Sensing Group at the University of Cape Town has been using Commercial Off The Shelf (COTS) machines for parallel processing. The *GOLACH* processor is a network of standard Pentium II machines running the *Beowulf* parallel processing software. The parallel processor is used primarily for the processing of data from various radar projects. Various MSc projects have been run to develop software and algorithms for processing on the GOLACH processor.

What is been investigated for the long term is the development of an embedded parallel processor that will enable more flexible use of parallel processing such as in aircraft. This thesis aims to investigate the requirements of a node for such an embedded processor and to investigate the use of software configurable hardware to perform application specific tasks at very high speed.

1.1 Project Objectives

This dissertation project is primarily the development of a hardware design to create a platform that will enable reconfigurable hardware parallel processing research at UCT. The specific objectives of the project were to:

1. Research existing parallel processing hardware and evaluate the strengths of the various architectures. Additionally, a review of the current use of reconfigurable logic in processing and parallel processing should be undertaken.

2. Devise a plan for developing a more general-purpose hardware reconfigurable processing node. This involves evaluating the various existing processors and selecting core components and where possible employing original ideas.

3. Design and implement the prototype hardware of a node for an isotropic parallel system.

4. Develop system software and firmware to enable the hardware to run application software.

5. Demonstrate simple examples of the hardware's capabilities and verify its operation.

6. Keep the cost per node low enough for the system to be viable compared to other systems.

1.2 Outline of Dissertation

The following chapters of this dissertation are structured in the following manner.

Chapter 2 provides a theoretical insight into the subject matter of this dissertation. A general overview of parallel computing is given. It provides specific emphasis on various parallel processor architectures in existence. A review of some existing reconfigurable parallel processing nodes is also provided. Following this is an overview of some of the various common methods used to network nodes of a parallel system together. Both the physical topologies and protocols used are discussed. Finally, a section on reconfigurable logic discusses the benefits and problems that these devices can have. A discussion of the practical limitations for implementing algorithms in reconfigurable logic is provided.

Chapter 3 provides the requirements and specifications of the project. The *user requirements* describe what the final system should be capable of delivering. Project deliverables are a summary and interpretation of the user requirements setting out definite goals for the project. The requirements analysis takes the requirements and deliverables and discusses how each requirement can be achieved. Lastly, an acceptance test specification is drawn up which provides a set of tests which the project hardware and software must pass to in order to show achievement of the requirements.

Chapter 4 describes the concept study performed before the commencement of the project design and implementation. It begins with a study of the requirements of a node for a parallel processor. This looks at the core elements required of a processing node to perform calculations as well as communicate with other nodes. Following this is a requirements review for the processor to be chosen is described. Various available commercial off the shelf (COTS) processors were investigated. A processor device was selected that was the most suitable and practical choice for the project. A study of the requirements of the reconfigurable logic described the various available choices and selects the one most suitable. A high-level system design is then formulated specifying the basic modules and their interconnections from which the design would consist. The choice of networking topology is then discussed and the various methods available were analysed to decide on the most suitable networking configuration for this project. Finally, the software and firmware requirements are discussed starting from low-level logic requirements up to parallel system software.

Chapter 5 describes the hardware design and implementation processes. This includes high-level conceptual designs and low-level system design diagrams and choices, component research and selection, as well as the implementation process. The implementation of every module of hardware is and their interaction with each other is described.

Chapter 6 shows the steps taken to verify the functionality of the system. Each individual component in the system is tested according to the acceptance test specification and requirements. A description of the results of the various firmware codes written in VHDL and schematic entry as well as software written in predominantly assembly and C as part of the acceptance test procedures is provided.

Chapter 7 describes the process of development of the firmware for the various configurable logic devices employed in the project. Each firmware design is presented with high-level overviews, detailed design specifications and discusses implementation details. Details of particular problems and their solutions are provided.

Chapter 8 discusses the design, development and implementation of software for the node microprocessor and host systems. The chapter firstly describes the design of software for the host system that interfaces with the project hardware. The various development tools that were used are discussed and the implementation of various utility software programs is described. Secondly, the design and implementation of software for use on the thesis hardware itself is described. This shows the development cycle of software for the project, starting with utility programs needed to initialise, control and test the system. This is followed by the implementation of an operating system to manage the hardware environment which allows the development and use of application software to execute without modification on the node hardware.

Chapter 9 describes the testing procedures, verification and results from the final implemented system. The algorithms and software used to benchmark the performance of the system are laid out. The benchmarks are compared with other standard processors to evaluate the relative performance of the hardware. Verification procedures for testing the validity of the results are also provided to prove the correct operation of the hardware. The results are compared to theoretical predictions of the hardware performance and a discussion of the analysis is given.

Chapter 10 gives the conclusions as to the success of the project. Recommendations for further work and suggestions for future hardware designs are provided.

Appendix A Schematics and PCB. These are the designs for the Node developed in this dissertation.

Appendix B Oscilloscope Traces. Various captures from the digitising oscilloscope.

Appendix C Propane system interface design specification.

Chapter 2

Theoretical Background

2.1 Parallel Computing

This section gives a brief background introduction to parallel processing and the core components required in any successful system. It will also highlight various techniques used especially in embedded parallel processing which is the focus of this project.

2.1.1 Introduction to Parallel Processing

There are two main types of multiple processor configurations used today [1,sec1.2]: Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). SIMD processors are usually highly customised and difficult to design and are not widely used. They perform operations in parallel by performing the same instruction on a number of parallel data inputs producing a set of parallel outputs. MIMD processors are more common. They typically consist of multiple processing units each running separate instructions on separate data. MIMD processors are further divided into two areas, Shared Memory systems and Distributed Memory systems.

Shared memory processors exist when multiple processors are linked together on a local memory bus and all have access to the same memory space taking care not to simultaneously access regions of memory in such a way that errors would occur. Distributed memory parallel processors have individual processors each with their own memory. Each processor has its own program to execute and a communications channel links the processors. For one processor to access the memory of another, messages must be passed between the two processors.

Distributed Memory Parallel processing is essentially the use of more than one processing unit linked via a communications network to perform processing tasks that would otherwise not be possible with a single processing unit. The advantage of this over a shared memory system is that tens, hundreds, or even thousands of these processors can be connected to communicate and compute together. Each processor in a distributed system is called a node and is capable of running independently of the

others. These nodes may also run their own operating system with multi-threading allowing multiple programs to run on each node.

On a distributed Memory Parallel processor, the communication between nodes is normally done by a set of libraries that work together to create a parallel virtual machine. The processing power of the system is dependent on the communications interconnects on the system as well as the parallel algorithm used. A good algorithm will keep message passing to a minimum as the processor can be kept waiting unnecessarily for data to be moved. Also, for a system with a relatively large amount of nodes, a 100% parallelizable algorithm needs to be used according to Amdahl's law [1,sec1.4.1]. Amdahl's law states that the maximum speedup is limited to the serial fraction of the program.

2.1.2 Parallel Software

There are two major types of software for parallel processing, Shared memory models for shared memory computers and message passing libraries for distributed memory system which are however also used commonly on large shared memory machines like the Cray T3D. The message-passing paradigm is the dominant form of software currently in use and matches the architectural design of a distributed computer.

The parallel computing libraries make the task of writing software for a parallel processor more focused on the algorithm so the user does not need to worry and the system topology or architecture. There are three industry standard libraries for parallel computing, MPI (Message Passing Interface), PVM (Parallel Virtual Machine) and SHMEM which is run on Cray supercomputers. Software such as PVM can be modified for a particular parallel processor simply by implementing the necessary low-level message passing mechanisms. The application software will be unaware of these implementation details.

When building a parallel computer, the systems programmers need to provide an operating system and extensions to a parallel processing library for the system. The most important part of the libraries is the communications subsystem, which need to use the hardware to its maximum.

2.1.3 Hardware Configurable Parallel Processors

The concept behind a hardware configurable parallel processor is that the electronic circuitry in a node can be reprogrammed to perform the currently required operation in the most efficient means. With the advent of advanced FPGA technology, these logic devices can be reprogrammed while running to allow the processing logic to be changed dynamically.

Presently most projects using configurable hardware such as FPGAs use them to create a configurable network or reconfigurable mesh. This allows the system to implement various interconnection topologies to optimise various algorithms and experiment with with arbitrary connection patterns.

FPGAs can also be used to implement algorithmic specific cores for each type of data processing required by the system. These can be performed many times faster than with standard general-purpose processors because it allows the programmers to optimise the circuitry and run multiple sub portions of an algorithm in parallel.

2.1.4 Communication Architectures

Various communication topologies exist for distributed parallel processing. Depending of the system size and algorithms run on the system, each one has its own strengths and weaknesses.

Important issues in parallel topologies:

Connectivity: The ideal situation is when the network if fully connected, thus every node has a direct link to every other node. This is impractical for even reasonably sized clusters.

Degree of connectivity: This is the the number of direct lines from each node, or the number of neighbours a node has. The larger this value, the quicker the communications, but it makes the software and hardware more complicated.

Static: A parallel computer is static when all the links are pre-defined and fixed.

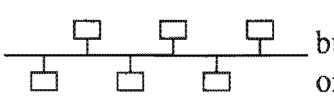
Switch: A node or set of nodes that only perform communications and no processing. A switch can be used to make the network topology dynamic or even fully connected. It can connect and communicate with any other node on the same switch directly.

In the case where a fully connected system is not possible or even in a small system designed to scale, a network topology for connecting the nodes must be used. There are many network topologies around and they all aim to either keep the system simple and cost effective or as close to fully connected as possible.

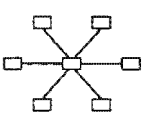
Bus type networks allow a type of fully connected system however all the communications are restricted to the bandwidth of the bus.

Direct one-to-one connected lines allow for the best speed and lowest latencies and allow the communications protocols to be simplified.

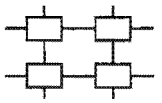
Some typical topologies are:



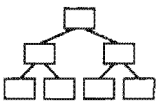
Bus: When a bus is used, it is normally a commercial bus like Ethernet and may not be very fast. However memory busses allow for very high performance but are limited to a few nodes. A cluster of PC machines on a network running a Beowulf system is a typical example.



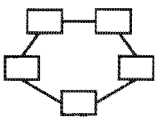
Star: Each node is connected only to one central node. The number of nodes that the central node can support is limited.



Array: A n-way array of nodes each with $2n$ lines to other nodes. Typically a 2-d or 3-d array. There are multiple paths from any given node to any other node. This is the basic structure used by the Intel supercomputers including ASCI Option Red (1.8 TFLOPS).



Tree: Nodes are arranged in a tree configuration. Each node has n children nodes and one parent node except the top node, which only has children.



Ring: Ring type structures can be formed by connecting opposite edges of an array topology together. The picture shows a ring from a 1-dimensional array.

2.2 Example Parallel Processors

Although many hundreds of parallel computers have been designed a build, very few have been designed to specifically exploit the potential of configurable logic to optimise the systems performance from a processing point of view.

Firstly, in an attempt to discover what traits a successful parallel processor has, some of the most powerful and successful parallel computers not using configurable logic are investigated.

The Intel Paragon

```
http://www.cica.indiana.edu/iu_hpc/paragon/pgon-tutorial
/section2_2_3.html
```

The Intel Paragon is a successful range of parallel processors that was reasonably cheap compared to the other commercial processors of the time. The processing nodes are in a 2D matrix arrangement of nodes connected to a backplane for communications. Each nodes runs a minimal operating system that performs the message passing and thread management. Specialised nodes on the edge of the array perform file access. The system also has a single applications processor that provides the user interface to the machine. Each node has two CPUs, one to run applications and the other is a dedicated communications processor.

Cray T3E Multiprocessor

```
http://www.cray.com/products/systems/crayt3e/
```

The Cray T3E is a scalable shared-memory processor with nodes containing COTS processors connected together in a 3D torus. Each node interconnect carries up to 480MB/s of data, which supports the shared memory system. I/O access is via the GigaRing channel provides 267MB/s of bandwidth for every four nodes. The T3E nodes incorporate advanced cache techniques to hide the memory latency of the system and increase performance. To take advantage of these caches, special optimisations had to be incorporated into the compilers to take advantage of the system.

Cray T90 and other vector processors

The vector processors use custom CPUs that provided multiple floating-point operations per clock tick. These systems are true shared-memory processors with each node connected to the main system memory with an extremely high-speed memory interface. Each CPU is synchronised by a central clock distributed via fibre optics. Each shared memory system is considered a node and multiple nodes can be connected via the GigaRing system.

The ASCI Option Red Supercomputer

`http://www.cs.sandia.gov/ISUG97/papers/Mattson/OVERVIEW.html`

Timothy G. Mattson and Greg Henry, Intel Corporation

This super computer was developed as the first in a line of super computers for the US Department of Energy (DOE) that had needs far greater than the then current fastest supercomputers. Intel was challenged to build the world's first and currently only > 1TFLOPS (Trillion Floating point Operations per Second = 1,000,000,000,000). It utilises a 2D mesh interconnection structure controlled by custom mesh routing chips providing four simultaneous 200MB/s channels to every other mesh routing chip. A network interface chip on each node connects to a mesh routing chip and sets up a route between two NICs though the mesh network. Each node in the system has two processors and a PCI bus to which COTS PCI cards such as RAID, ATM and FDDI cards are connected. Various nodes in the system also run specialised operating systems depending on their function. The entire system is designed with redundant components to allow system operation to continue in the presence of hardware failures. This system used no reconfigurable logic and all processing was done in COTS processor chips.

Kendal Square KSR1

`http://wwwmcb.cs.colorado.edu/home/capp/ksr.html`

This is an older highly parallel system using hardware supported "distributed-shared" memory. They use a caching technique they call ALLCACHE memory that allows

each CPU to reference any memory location in the machine. The local memory then becomes a cache of another memory location. A cache-miss will prompt the hardware to search first locally and then distributed for the location of the addressed memory. The search hardware is called the search engine and it runs on a custom set of rings and directories for finding and moving memory pages.

Huinalu Linux SuperCluster

`http://www.mhpcc.edu/doc/huinalu/huinalu-intro.html`

This machine is the latest in a stream of new highly parallel machines based on COTS components. This machine has 260, dual P3 933MHz nodes using standard IBM rach mount computers. The theoretical maximum processing speed is 478 GFLOPS. The general-purpose components make this machine cost $1/10th$ the price of an equivalent custom supercomputer.

CM-5

`http://csep1.phy.ornl.gov/cm5_guide/cm5_guide.html`

The CM-5 contains a large set of processors divided into groups. Each partition has its own processor, a partition manager. Each processor is a SPARC based processor with four vector units in parallel. The whole arrangement of nodes is a distributed memory system.

There are two communications systems in the CM-5. A data network and a control network to which all nodes are connected. The control network is used to synchronise nodes and perform global operations. The data network is used for inter-node data communications at 20MB/s.

This machine supports a maximum of 16,384 nodes giving a theoretical maximum speed of 1000GFLOPS

Fujitsu VPP Architecture

`http://www.fujitsu.co.jp/hypertext/Products/Info_process/hpc/vpp-e/index.html`

The Fujitsu high performance computing machines are Vector Parallel Processors (VPP) based on custom LSI CMOS devices. Each processor in the VPP5000 for example performs over 8700 MFLOPS for the LINPACK benchmark. Each processing element contains up to 16GB of memory with the total system containing up to 2TB.

The processing elements communicate on a crossbar switch supporting 1.6GB per second in two directions simultaneously. The system runs a Unix operating system and provides distributed parallel filesystems and high speed networking interfaces.

The maximum theroetical speed is rated at 1228 GFLOPS on 128 processing elements.

2.3 Example Reconfigurable Processors

Armstrong III

`http://www.lems.brown.edu/arm/`

The Armstrong III processor is a 20-node parallel computer build to research merging a processor and reconfigurable logic device in each parallel node. Each node has eight high-speed links, which can allow the system to be arranged in numerous configurations. Each node contains a communications board and processor board. The comunications board contains a dedicated communications processor and communications FPGA. The processor board contains a RISC processor and FPGA as well as memory and serial ports. The system showed that the FPGA devices greatly accelerated the performance of the system.

ArMen

`http://ubolib.univ-brest.fr/~armen/armen1-eng.html`

The ArMen processor is a parallel processor with each node containing a FPGA coprocessor. Each node has up to 4Mb of ram and has a T805 processor with four 20Mb/s links. The interconnect architecture is configurable to optimise various applications requirements.

2.4 Reconfigurable Logic

In many new high speed computation and communication systems, the flexibility requirements of the systems are becoming more in demand. The flexibility in the past has been mostly concerned with the software because this was the only component of the system that was modifiable. With the advent of a new-generation of FPGA devices that support run-time reconfiguration, the scope of reconfiguration in a system can now include logic firmware flexibility.

Firmware re-configuration of the logic in an FPGA device allows system designers to implement systems requiring optimised high-performance as well ad flexibility. Cost savings can also be achieved by using a single device for multiple functions, selecting an appropriate configuration when required.

The current major problem with reconfigurable logic is that there is a major lack of design tools to enable run-time reconfiguration. Most designs implement static configurations that are loaded into the device, but as the development software evolves to catch up with the hardware, designs that reconfigure partially without interrupting other parts of the same device will be simpler to implement.

Chapter 3

User Requirements and Specification

This chapter develops the requirements and deliverables for the project as well as a specification for the project hardware and software design. No past research into hardware reconfigurable parallel processors has been conducted at the University of Cape Town, and thus the requirements and specifications are largely drawn from the basic requirements and research on other existing reconfigurable and parallel processing systems.

Firstly, a user specification is provided which specifies the high level objectives of the project. These specifications make no reference to specific hardware or software usage. The project deliverables are then developed from the user requirements and are followed by an analysis of the user requirements. A system specification is then developed followed by a set of test specifications to which the implemented system must comply.

3.1 User Requirements

The development of a software and hardware configurable parallel processor node has the following functional requirements:

1. The system must demonstrate the principle of using a configurable logic co-processor to implement common functions such as the FFT algorithm for use in radar processing, or the Discrete Cosine Transform for use in image processing. The system must also demonstrate the parallel execution advantages of configurable logic.

2. A node shall be a low cost and power efficient processing module that can run as a stand-alone processor. All the nodes in the parallel system will be identical from a hardware perspective (isotropic).

3. Each node must contain a communications unit, central processor, and configurable logic unit capable of interfacing with the processor.

4. Each node must provide support for high-speed inter-node communications.
5. Each node will run its own copy of an open-source operating system and parallel processing libraries. The nodes will form a MIMD processor.
6. A minimum of two nodes must be built to demonstrate the nodes performing parallel processing functions.
7. Speed comparisons of various networking topologies must be performed for common parallel algorithms.

3.2 Project Deliverables

The fundamental goal of this project is to produce a working node for a Parallel Processing Unit (PPU) and to demonstrate a simple, hardware accelerated parallel algorithm running on the node. The nodes are to be comprised of a configurable logic unit and a general-purpose microprocessor with supporting hardware. Each node is required to run an open-source operating system and demonstrate a parallel processing example.

The system is to be configurable in software and hardware. Software configurability means the ability to change the algorithms and software running on the system. Hardware configurability is the ability of the system to change the behaviour of certain logic components to optimise the system for specific processing needs.

As secondary goals, the following deliverables should be attained: The implementation of networking protocols across the configurable communications infrastructure. The use of configurable logic for optimising various computations.

As tertiary goals, dynamic re-configuration to constantly optimise the system performance and dynamic network configuration should be investigated.

The functional hardware platform is the major deliverable. Software for the processors, configurable logic, communications interfaces, communications libraries and parallel processing libraries are also part of the deliverables.

3.3 Requirements Analysis

After considering the requirements, it is clear that no specific hardware design constraints are implied other than providing the required functionality. Therefore after an analysis of typical microprocessor systems and the requirements of a node in a parallel processor, the following more focused requirements have been developed.

3.3.1 Silicon requirements

The basic component requirements of the reconfigurable parallel processing node are illustrated in figure 3.1.

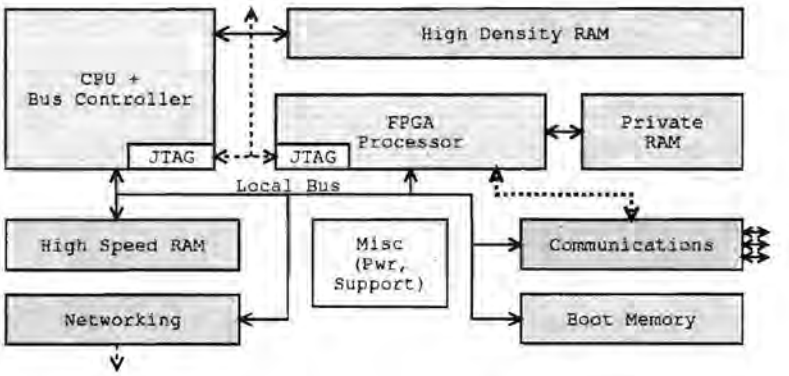


Figure 3.1: Basic Component Requirements of a Node

1. Processor, a high performance, low power embedded processor with the capability to interface with a: Communications controller, FPGA and Memory Infrastructure.

2. A high gate count FPGA with an interface to the processors memory bus to provide high-speed access. Advanced clock generation would be beneficial for implementing high speed designs.

3. A low gate count communications FPGA with high speed interfacing technologies. The most commonly provided interfaces being Low Voltage Differential Signalling (LVDS) or PECL.

4. Ethernet like communications infrastructure capable of at least 10Mb/s and possibly 100Mb/s speeds.

5. Non-volatile boot memory for standalone system operation.

6. JTAG Test and Access port or similar debugging and testing interface.

7. Serial communications (EIA-232) for system control and debugging.

8. High-speed inter-node communications infrastructure support.

9. Power supplies and system control components.

10. Processor/system boot-up configuration if necessary.

11. Status/debugging indicators.

3.3.2 Mechanical requirements

Ball Grid Array (BGA) and other high density chip packaging products are to be avoided where at all possible to reduce system cost and reduce the risk of manufacture or design error. For future designs, this limitation may be removed.

The physical dimensions and interfacing connectors must comply with the requirements of the backplane or system architecture to which the nodes will interface, if any. This will be specified in the specification and hardware design chapters.

3.3.3 Interface requirements

1. Ethernet or some other high-speed means will provide communication and software downloading to each node.
2. Serial communications must be used for debugging and console use.
3. FPGA communications must use high speed (>100Mb/s) LVTTL, PECL or LVDS signalling between nodes.
4. FPGAs must be memory mapped to the processor and provide an interface to processing and communications logic.
5. Comms FPGA should have bus mastering or DMA capabilities for fast memory transfer operations.
6. Parallel processing libraries will provide a technology independent interface for messaging between nodes.
7. The operating system must provide a TCP/IP link over a custom communications channel to allow transparent simple communications between nodes.
8. For the prototype hardware design developed in this thesis, an interface to a standard PC is required, possibly across the PCI bus.

3.3.4 VHDL / Macro-function requirements

1. Communications controller.
2. Coprocessor functions.
   Including an FFT or DCT processor
3. System bus interface controller.
4. Advanced clock generation and deskewing.
5. RAM buffers for communications and processing.

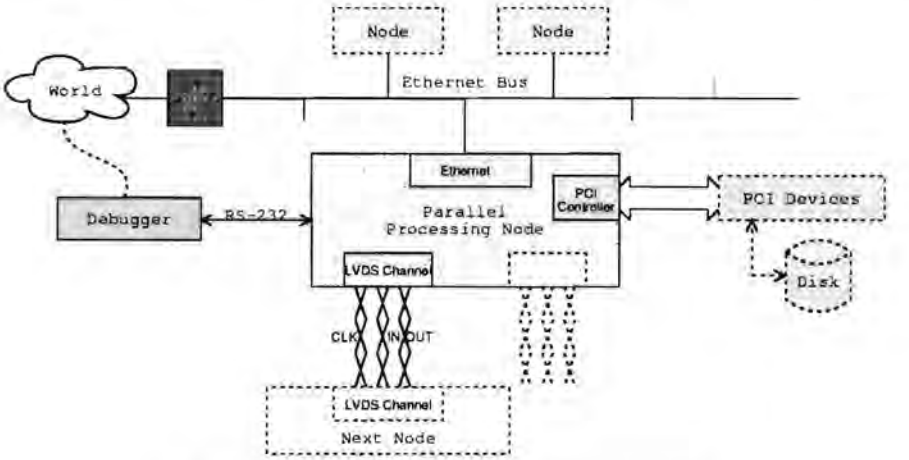


Figure 3.2: Node interface requirements

6. FPGA configuration support

7. System boot and configuration

3.3.5 Software requirements

1. Operating system (Linux or eCos) ported to a node.

2. Ethernet or equivalent networking: Programming, control, messaging.

3. FPGA run-time reconfiguration if FPGA supports it.

4. Interfacing with FPGA processor and communications.

5. Parallel processing libraries ported to OS/communications network.

6. Test suites.

3.4 System Specifications

3.4.1 Processing Algorithms

1. FFT or DCT algorithm demonstration.

2. Parallel execution of algorithms.

3.4.2 Communications

1. GDB / Custom debugging interface via RS-232 channels.
2. TCP/IP Stack over Ethernet or equivalent.
3. TCP/IP Stack on Custom Network Mesh.
4. Configurable inter-node topography.

3.5 Test Specifications

The system will initially be tested with a single node in order to develop the VHDL firmware and port and test the operating system to the platform. Acceptance test procedure tests will be developed to verify the functioning of the hardware and firmware designs. Testing will be performed using the serial port and with test software in the FPGA. If a direct interface to a PC system is implemented, this interface can be used to test system components without the functioning of the processor.

Later testing will be done at a higher level over a communications link and will provide access to features such as a remote shell if the Linux operating system is used and configuration such as programming the flash and setting up the software on the node.

Final testing will test processing algorithms running on the processor and FPGA, verfiy the correct operation of those algorithms and finally demonstrate a parallel processing example.

Chapter 4

Concept Study

The idea of a hardware and software reconfigurable parallel processor is not a novel idea, however not many such systems have been developed. In most cases, the projects have been either commercial or research using proprietary technology. This makes open research in this area more difficult due to the lack of information provided and the lack of physical hardware.

A hardware configurable processor with the use of open-source technology hopes to further the use of open-source software for parallel computing and to develop an environment for further development and research. This chapter describes the concept study undertaken at the beginning of the project in order to determine the viability and further develop the requirements and specification of the system.

Firstly, the requirements of the parallel node in general are discussed; this is followed by more detailed looks at the processor and reconfigurable logic. A high level system design is specified followed by discussion of the various choices for inter-node communications. Finally the software requirements for the system are expanded upon.

The results of this study were that the project goals of developing a node for configurable parallel processing were determined to be within reach using currently available technologies.

4.1 Parallel node requirements

Essentially, each node needs to be capable of running as a stand-alone entity without support from other nodes. This means that and as such, it will require some of the basic elements needed for a microprocessor system:

1. Processor: This can be either a stand-alone, System On Chip (SOC) or ASIC implemented processing device. This will provide the general processing and control requirements of the system.

2. Memory: Memory for system and application software as well as data storage is essential for the operation of a microprocessor. Types of memory that should be

considered.

(a) FLASH, a medium density non-volatile memory primarily for program and data storage.

(b) SRAM, a high speed memory low-density memory for low latency and high bandwidth data storage.

(c) DRAM/SDRAM, a high-density memory type with reasonably high speed for program execution and high volume data storage.

3. Communications. In order for a node to communicate with other nodes, various forms of communication may be required.

(a) Ethernet networking is a well-defined industry standard communications bus, with average bandwidth capabilities.

(b) Backplane communications provide very high-speed connections but can only support a limited number of nodes.

(c) Custom communications can be provided though link layer chip-sets or implemented using configurable logic devices. They have the potential for producing very high speed point-to-point or bus type connections.

(d) Standard Serial. These low data rate communications may be used for control and system configuration as well to aid in debugging.

4. Power supplies for the various system components and microprocessor supervisory, control and configuration devices.

5. Hardware interface devices and physical connectors.

To support the need for reconfigurable processing, a configurable logic device with enough resources to support the algorithms required will be needed.

4.2 Processor requirements

The scope of this project is to develop a basic node for parallel processing and the focus is more on the design than absolute processing speed. Where possible, devices such as Ball Grid Array (BGA) will not be used because of the added design and manufacture problems.

4.2.1 System Requirements

The major requirement of a processor for the parallel processor is CPU processing performance. The major processing tasks of the CPU will be communications and integer/floating point mathematics. DSP instructions may be used for particular processing algorithms but these will usually need to be routines written in the processors assembly language and very carefully optimised. The Multiply And aCcumulate (MAC) features of some processors can also greatly speed up certain algorithms.

The processor (with companion memory controller if necessary) must be able to interface to SDRAM, SRAM, FLASH and the FPGA preferably with a glue-less interface. This will mean that the hardware design will be greatly simplified and the risk involved with interfacing to complicated devices such as SDRAM will be reduced. The ability to perform Direct Memory Access (DMA) transfers on the bus will allow the FPGA and processor to use a shared memory arrangement which will reduce the overhead of copying data through the CPU.

4.2.2 Processor Selection

The MIPS range of processors (see [2] for MIPS information) generally come in small easy to use packages and have very high performance. The MIPS instruction sets are supported by Linux, eCos and the various 'BSD open-source operating systems and most importantly the GNU C Compilers (GCC). A port of an operating system to the project hardware will be greatly accelerated by using existing ports to the same processor architecture. In general, most mid-range MIPS processors come in a low pin count package configurations and have an interface to a support chip (companion) for memory bus and peripheral bus access.

The IDT 79RC64574 [3] processor is a 64bit MIPS processor capable of running up to 333MHz with a maximum performance of 444 Drystone MIPS (Millions of Instructions Per Second) in a 128pin PQFP package. This processor features a double precision floating-point unit running up to 666 MFLOPS (Millions of Floating Point Operations Per Second). It also includes DSP extensions for up to 125 million multiply and accumulates (MACs) per second, a full-featured virtual memory manager and 32kb data and 32kb instruction caches. The '574 has a 32bit wide external data/address bus (SysAD bus) and the '575 is 64bits wide. Other variations of the 64bit MIPS processors are available from IDT, NEC and Toshiba all supporting the same instruction sets and similar bus interfaces. The GNU C compiler supports the full range of standard MIPS processor instruction sets. The internal architecture of all MIPS processors are very similar internally and are backwards compatible which makes migrating to newer devices very easy.

The embedded PowerPC chips from IBM and Motorola run up to 550MHz and support a host of on chip peripherals. These devices typically come in the 300 to 600 pin BGA package types and are very complicated to design with. These devices are typically aimed at communication processors and support a host of interfaces and

features. Most of the features provided are not needed for a parallel processing node. The GCC compilers and most open-source operating systems support the PowerPC range of devices.

The Hitachi Super-H processor range SH-4 runs up to 200MHz delivering 360 Drystone 1.1 MIPS. It comes in a 208 pin QFP or 256Pin BGA. The SH-4 has added instructions for vector manipulation mainly aimed at graphics processing which could be used for accelerating certain algorithms. There is not a wide range of these devices available and backward compatibility and future support for these processors is questionable.

Intel's Strongarm processors run up to 235 Drystone MIPS at 206 MHz and come in a MicroBGA package. The are supported by the GCC compilers and are based on a modified ARM core. The number of devices available is limited and future continuation of this range of processors is questionable.

Overall, the MIPS processors have the greatest performance to complexity ratio of all the processors evaluated. The are available in easy to use packages and there are many pin compatible chips available from multiple manufacturers. The processors are designed to be dedicated processing units and added peripherals and interfaces are available on a range of MIPS companion chips. Linux, eCos and BSD's support the MIPS processor range, which will greatly aid in the porting of the chosen operating system.

4.3 Reconfigurable logic requirements

For an FPGA to be used as a viable device for implementing arbitrary coprocessor functions, the device should have the resources to support any synthesisable design up to some limit. In certain cases, a parallel implementation of the algorithm itself on a device can be created if the resources are available. This can allow each node in the parallel system operate as internally parallel in addition to the parallelism of the system. There is also the need to possibly implement a reduced CPU core on the FPGA device. This could allow more complicated processing functions to be implemented on the FPGA and reduce the amount control required from the microprocessor. The implemented CPU could be used to feed data into coprocessor units and write results back into a shared memory as an example. This CPU could also be used to perform communications functions and provide intelligent switching of the inter-node communication links.

A device in the order of a 1 million gate FPGA will provide the flexibility to implement a very wide range of functions and provide the resources to implement multiple copies of simpler functions to create a super-scalar system. However these devices are very expensive and a lower gate count device should probably be used for the prototype implementation.

To experiment with high speed interconnects; the FPGA used must support high speed I/O such as LVDS (Low Voltage Differential Signalling) serial channels, which

can run up to 840Mb/s in some FPGAs. The FPGA-CPU interface must be capable of handling the full burst rate of the bus. This should allow maximum data throughput across this interface.

The Xilinx Virtex-E FPGAs [4] support multiple (limited by device pin count) 622Mb/s LVDS channels running off a single clock channel. They do not provide any hardware serialiser/deserialiser (SERDES) circuitry however the application notes provide reference designs to implement these in low-level components. All the Xilinx Virtex-E devices provide LVDS support.

The Altera APEX20KE/C devices support up to 16 622Mb/s channels on predefined pins. Only devices in the KE and KC range with greater than 400,000 gates support LVDS. These devices do however provide hardware SERDES units however lack flexibility in their use.

4.4 High level system design

The structure of each node in the parallel processor will be identical from a hardware perspective. At some stage, certain nodes in the system may be required to perform different tasks. Therefore, the design of a node must make provision for general purpose computing functions in addition to the hardware reconfigurable processing units.

Each node will contain a MIPS based processor preferably an IDT or NEC MIPS R4000 or R5000 64bit device. These devices can double the integer performance of 32bit processors when used correctly under certain conditions. The IDT processors also provide a high speed floating point unit that operates in parallel with the integer unit and employs dual issue support to create a scalar pipelined architecture. This means that the processor can execute a single integer and floating point instruction simultainiously. These devices also provide multiply and accumulate and other DSP extensions which can be utilised. These processors also all have advanced memory management, which support advanced operating systems like Linux and BSD. This can be used to create a totally hardware independant environment for applications to execute in which means that they can be very easily ported to other hardware platforms. Moving between MIPS platforms will in most cases not even require a recompile.

The IDT MIPS processor requires an interface chip to provide memory interfaces and peripheral bus support for the system. The interface chip performs the interfacing and controlling of the various memory components without processor intervention.

The interface chip will connect to at least 16MB of SDRAM memory possibly supporting DIMM format modules or on board devices. There will also be a 16Mbit of Boot FLASH memory for boot loader and operating system storage.

A Xilinx FPGA will be connected with an SRAM or similar interface (configurable) to the memory bus. The Xilinx FPGAs provide a greater flexability for design and support LVDS channels in all the Virtex-E devices.

The FPGA will have a high-speed SRAM device directly connected to it for private high-speed data storage. This removes the problem of having the FPGA compete for

the usage of the processor local bus.

The FPGA will provide at least four LVDS channels for point-to-point inter-node high-speed serial commmunication.

The low-speed memory devices (FLASH) may need to be located on a peripheral bus to reduce the loading of the local bus.

The prototype hardware platform will interface to a standard PC system via the PCI bus. This will allow the devices in the system to be accessed transparently from the host PC which will greatly aid in development, debugging and also provide communications capabilities. A node may access devices on the PCI bus in the system if it provides bus master support. This for example will allow nodes to access standard PCI ethernet controllers. At a later stage, the nodes could be connected to a passive PCI backplane for standalone operation. Because the nodes will interface to a PC system, they must conform to the specifications of the PCI bus.

4.5 Communication infrastructure choices

The various parallel processor systems examined during the pre-study use a variety of different networking topologies. Some were limited to the system hardware capabilities while others implemented complex communications processors. By using a configurable logic device, we have the freedom to implement a wide variety of communications structures if the basic physical interconnects are well planned.

The various options for a network topology are as follows:

1. A serial chain or loop of nodes. In this system, each node will contain two links to two opposite neighbours. This system has the advantage that switching and communications software can be implemented in a very simple manner. A message to any node can be broadcast in both directions (or one direction in a loop) and be passed from node to node until the addressed node receives the message. There is however a very low level of connectivity and the total communications bandwidth is limited to the bandwidth of one link.

2. A square matrix of nodes. This is a common structure and is most notably used by the Intel Paragon supercomputers. Each node has a link to its neighbour on four sides. Messages are routed though nodes using routing algorithms. The total bandwidth of the network is much lager than the chain topology as there are multiple paths between any two nodes and a message only uses one path, leaving the other paths free for other nodes.

3. Cubic and hyper-cube structures are also possible, however the hardware and software becomes increasingly more complicated.

4. A Tree structure. These types of networks are useful when implementing an algorithm where jobs are dispatched from a central node and all results are returned back to that node. No inter-node communications other than with the

master node is performed. In this situation, the number of paths from the master node to the furthest node is minimised for a node with a fixed number of interconnects.

5. A star network. These networks are normally made of non-symmetric nodes with a central communications node having much greater IO capabilities to the rest of the nodes. The aim of this project however is to develop a low cost system of generic nodes.

6. A Bus network. Each node will have support for Ethernet networking which is a bus structure. Thus a bus structure need not be considered for the custom topology. The Ethernet bus may be switched to make the system act as if it was fully connected.

Based on these various structures, a square matrix of nodes configuration is simple to implement and allows for configurable topologies. This will give each node four bi-directional configurable ports for interfacing to other nodes. Each port may consist of one or more pairs of LVDS channel depending on the FPGA resources.

The configurability of the FPGAs will also allow for the nodes to be reconfigured in a ring or torus by linking the outer nodes of opposite sides of the matrix. The network can also be reconfigured in to a tree structure by changing the routing with each node supporting up to three children.

This structure can also be used to investigate using the FPGAs to switch traffic through one port and out another to provide transparent access between nodes and reduce message handling by each node between the two communicating nodes. This project however will focus on the hardware design and basic system functionality and these options will be left for future work.

4.6 Software requirements

The primary requirement for the system is to port an operating system to the hardware platform and to demonstrate the processing capabilities of a single node. Further work must demonstrate the ability for inter-node communications and demonstrate a simple example.

All the software required for this project is required to be built on open-source projects and tools. The development tools and operating systems selected will be open-source and most of the standard parallel computing libraries especially MPI and PVM already open-source projects.

The operating systems available for running on the MIPS platform are: Linux, eCos, NetBSD, L4, QNX

NetBSD [5] runs on a host of MIPS based machines and on the NEC and Toshiba based MIPS palm-top devices as well as some Silicon Graphics machines. The NetBSD kernel provides support for a host of platform independent architectures including PCI,

ISA, Ethernet, USB and other commercial busses. The strong point of NetBSD is that it has been designed specifically portability and runs on multiple architectures. It can also run Linux applications with the aid of a compatibility mode or layer. The GNU tools are the primary development bases used for the NetBSD.

Linux [6] has been ported to the MIPS processors in at least two forms, the standard kernel and real-time Linux. The Standard GNU tools are used to develop code for the Linux MIPS platform. Linux supports many of the R4000 and R5000 based MIPS chips except those with an on-chip L2 cache controller. The IDT MIPS processors do not have this feature and thus should pose little problems.

eCos [7] supports the NEC Vr4300 64bit MIPS processors. eCos however is a hard-real-time system and thus does not use memory management and is staticly compiled. This make changing applications or running multiple aplications simultainiously much more difficult.

The L4 microkernel is a real-time kernel and runs on the R4000 MIPS CPUs but a large percentage of it is hard coded assembler, which may prove difficult to port. (see [8]) User mode Linux can run under the L4 kernel, however for this project, a real-time system is not neeed and may prove unnecessary.

QNX is a non-opensource, free to use operating system that support the IDT MIPS processors. It therefore is not fitting of this project.

In general, NetBSD, eCos and Linux all use the GNU compiler tools, which provide generic support for the various MIPS versions. This means that the main porting requirements for these operating system will be the provision of devices specific drivers and boot up initialisation code.

Chapter 5

Hardware Design and Implementation

After the requirements analysis and specification was performed, the hardware design was developed. This chapter describes the hardware design process and the implementation of that design. The design process is presented starting from the high level design showing major functional units and their methods of interaction. Increasingly more detailed design descriptions of each functional unit and sub-parts thereof are provided. The entire design would be too large to cover in a single low-level description. Various difficulties and complicated design choices are explained in further detail where applicable.

The design of the system and each sub-section was specified to conform to the requirements and specifications laid out in Chapter 3. The design choices made are indicated with motivations for the various choices given. Implementation details are provided for each module to give an understanding of the low-level functioning of the components in the system.

5.1 Processing Node Components

After the review stage, specific processor, companion devices, memory and FPGA devices were chosen for the design. The major components needed chosen and sourced early in the project before the design had been finalised because of long lead times.

The processor chosen for the project was the IDT 79RC64754 64-bit MIPS RISC microprocessor [3]. This part was chosen for a number of reasons. It has 32-bit wide external data bus would make hardware design much easier and cheaper and reduce the risk involved in designing with a 64-bit bus. The device provides a double-precision floating point unit, which operates in parallel with the integer unit through a dual-issue architecture. Finally, the device is packaged in an easy to use 128-pin package.

The MIPS companion device selected for the project is the Galileo GT64115 [9]. This was selected primarily because it interfaces the same 32-bit SysAD bus as the IDT processor. It also has an integrated PCI bus master interface, which would make interfacing the PCI bus simple. An on chip SDRAM and memory controller allows

for a 'glue less' interface to SDRAM. To access SRAM and FLASH devices however, a bridge or bus decoder circuitry is required. Examples are given for designing such interfaces in the datasheets.

The FPGA chosen is the Xilinx XCV200E, 200,000-gate device in a 240-pin package. This device provides up to 64 LVDS pairs, 114k bits of block RAM and 20 I/O standards. The major factor influencing this choice was cost. Higher gate count FPGAs are available for later use but at a much higher cost.

Other core components required are SDRAM for main processor memory, FLASH memory for program storage, SRAM buffers and supervisory devices. Also essential are power supplies for the various components.

5.2 Design Choices

This section describes three possible system implementation architectures and lists the pros and cons associated with each. The final implementation details differ from these designs, which are from early in the design phase. They do however give an indication of the basic system architectures.

5.2.1 Choice A

See figure 5.1.

The FPGA is acts the bridge between the local bus and the peripheral bus.

Advantages

- High-speed operation with minimal loading on memory/data bus.
- Direct 32-bit interface to FPGA maximises data transfer rates and allows DMA access.
- FPGA can perform DMA transfers direct to SDRAM / PCI bus. This reduces CPU usage for data transfer and doubles memory bandwidth.
- FPGA and CPU ROM allows stand-alone system operation (Without host system intervention).
- SRAM cache memory for high speed FPGA reconfiguration.
- Partial reconfiguration possible via direct PCI I/O operations.
- Single CPLD for system configuration and FPGA configuration management.
- Simpler routing of PCB.

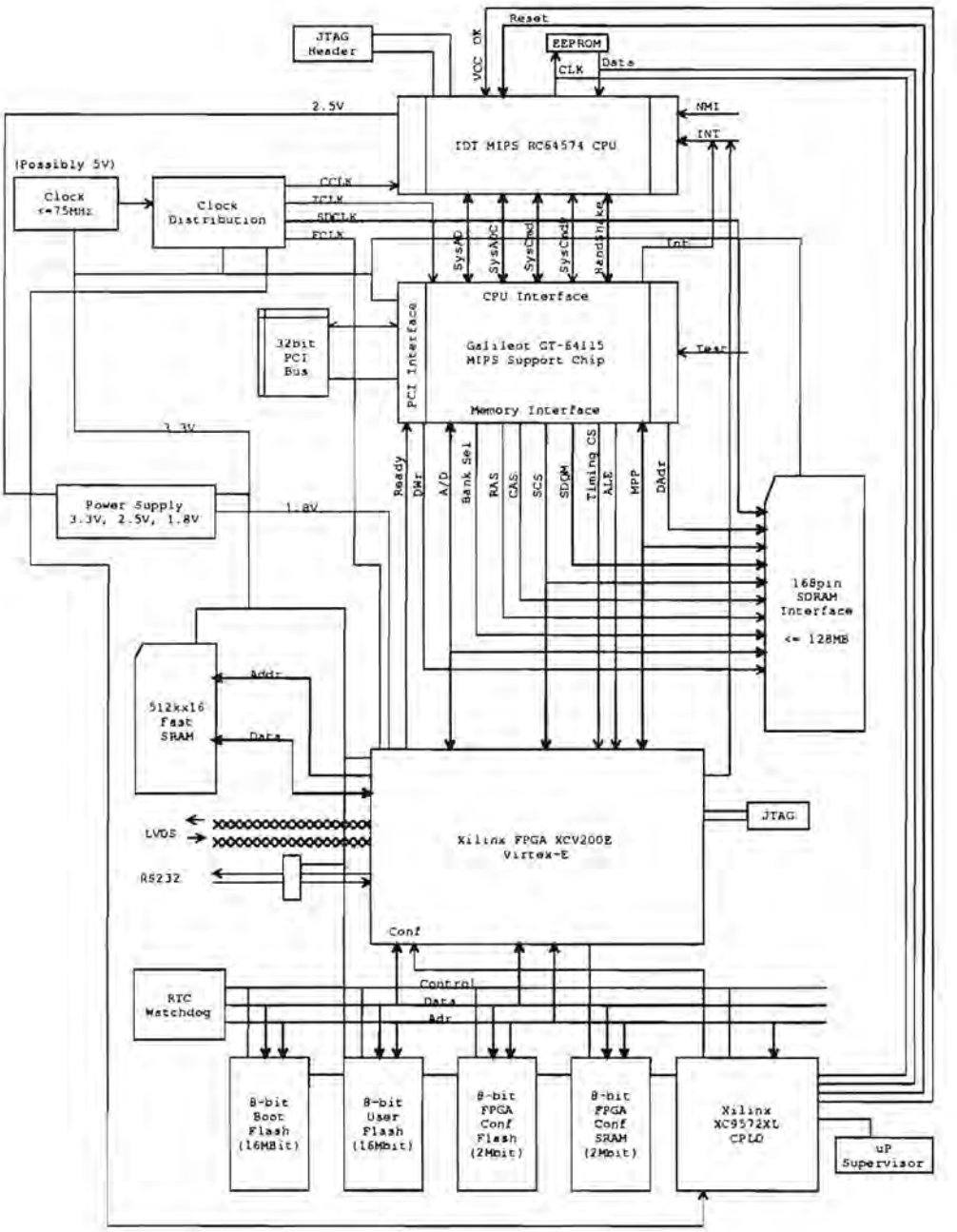


Figure 5.1: Design architecture choice A

Disadvantages

- FPGA requires a simple default configuration to allow access to CPLD and SRAM for reconfiguration from the CPU and PCI bus.
- Cannot perform direct I/O full configuration of FPGA, must be cached in SRAM first.
- FPGA configuration errors will prevent system operation.
- Peripheral bus operation dependent on FPGA operation.

5.2.2 Choice B

See figure 5.2.

The FPGA resides on the local bus along with a separate CPLD to act as a bridge to the peripheral bus.

Advantages

- High-speed operation with greater loading on memory/data bus.
- Direct 32-bit interface to FPGA maximises data transfer rates and allows DMA access.
- FPGA can perform DMA transfers direct to SDRAM / PCI bus. This reduces CPU usage for data transfer and doubles memory bandwidth.
- FPGA and CPU ROM allow stand-alone system operation (Without host system intervention).
- SRAM cache memory for high speed FPGA reconfiguration.
- Full/partial reconfiguration possible via direct/cached PCI I/O operations.
- Hardwired peripheral bus though CPLD device not dependent on FPGA operation.

Disadvantages

- Greater loading on high-speed memory bus.
- Two CPLDs may be necessary.
- More complex PCB routing.

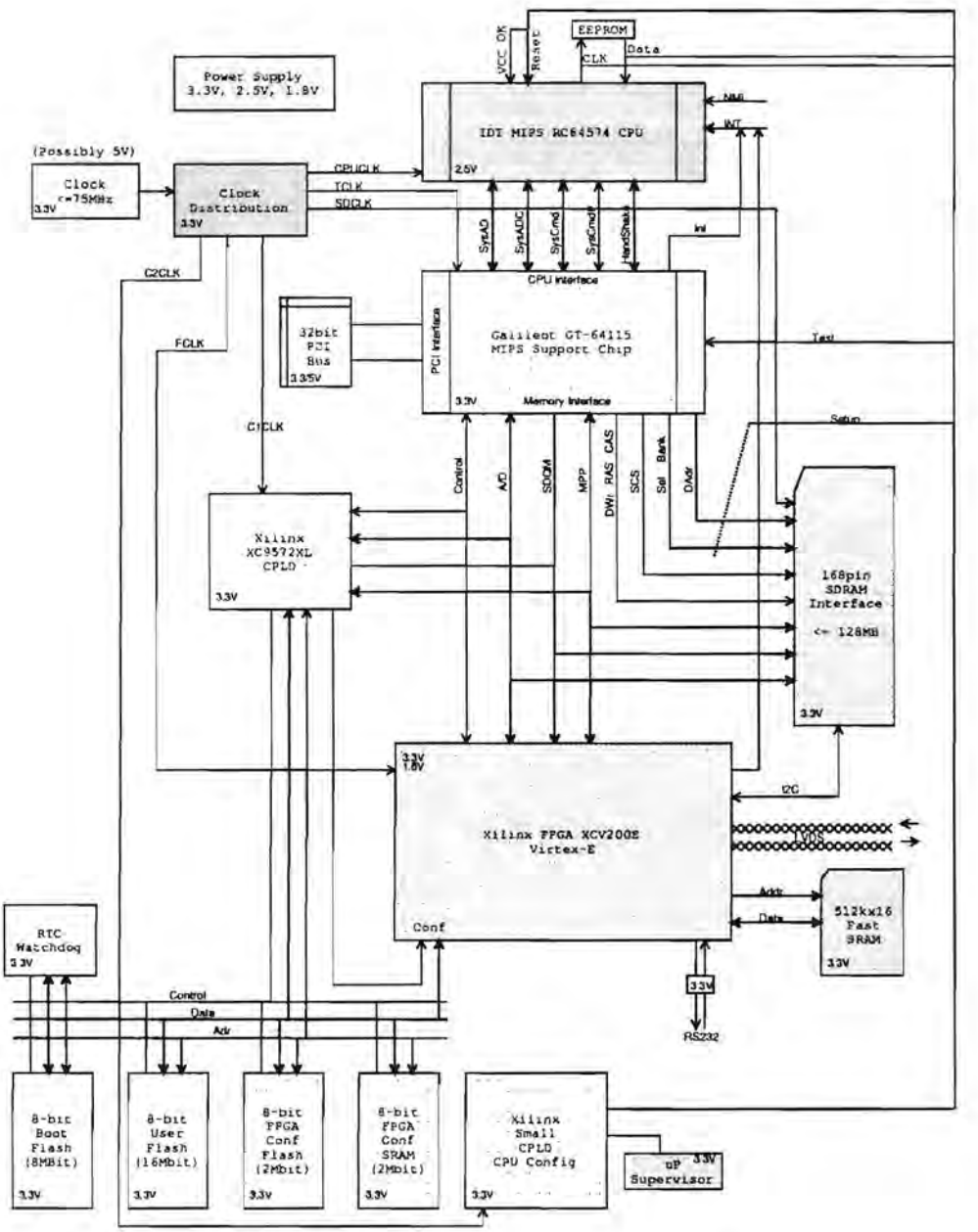


Figure 5.2: Design architecture choice B

5.2.3 Choice C

See figure 5.3.

The FPGA resides on the peripheral bus, with only a CPLD device and SDRAM on the local bus.

Advantages

- Reduced loading on memory/data bus.
- FPGA and CPU ROM allows stand-alone system operation (Without host system intervention).
- SRAM cache memory for high speed FPGA reconfiguration.
- Full/partial reconfiguration possible via direct/cached PCI I/O operations.
- Hardwired peripheral bus though CPLD device not dependent on FPGA operation.
- Simpler PCB routing.

Disadvantages

- Buffered access to FPGA reduces interface speed.
- Two CPLDs may be necessary.
- DMA operations may not be possible.
- Full bus width probably not available to FPGA, performance implications.

5.3 High Level System Design

The basic architecture from design option B in the previous section was selected for the design. The major design feature of this setup is that the FPGA is located on the local bus of the memory controller along with a separate 'Bus CPLD' that will act as a bridge to the peripheral bus as well as providing the logic required to configure the FPGA.

The MIPS CPU to be used requires various interfaces to be provided for operation. Most importantly, the processor bus (SysAD bus) needs [3, 15-1] to interface with a memory controller. The GT64115 controller provides a directly compatible interface, which will serve this function. The processor requires various interrupt inputs, which are provided by the various active components of the system. All possible future interrupt sources must be accommodated for in order to make the design more

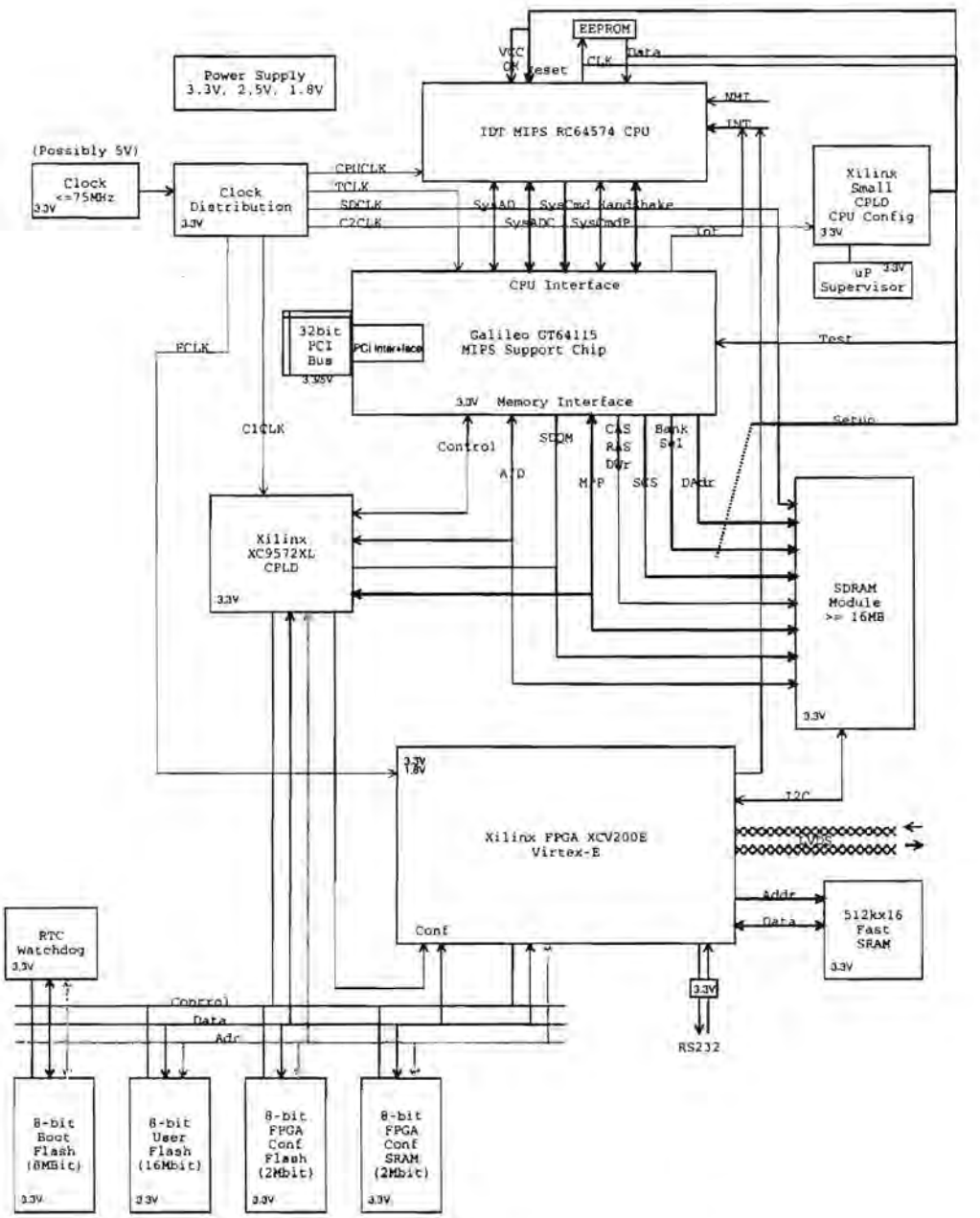


Figure 5.3: Design architecture choice C

generic. The configurable logic devices provide further possibilities. The processor also requires a configuration bit-stream which must be provided by either an EEPROM memory or dynamically generated. This will be one of the functions provided by a 'Config CPLD'.

The memory controller does not require any interfaces its self, it merely provides functions for the requirements of the system. The need for high speed high-density memory is satisfied by inexpensive SDRAM and the memory controller provides an interface for controlling this memory.

The FPGA will have a direct connection to the local bus. By providing as many bus signals as possible to the FPGA device, the configurability and options available for implementation is kept high. It also introduces the possibility for the FPGA to emulate the memory controller if it has access to all the required signals. The FPGA has the following modes of configuration: JTAG, Serial bitstream and 8-bi Parallel. JTAG programming is very simple and works reliably with a simple JTAG programmer. Support for JTAG programming will be provided for initial system testing and as a backup in the event that other methods do not work. Serial configuration requires a special protocol or serial memory device to program the FPGA. The FPGA's parallel programming interface is the fastest method available for configuration. The interface is 8-bits wide and can interface a standard memory bus with some additional signals for the controller of the bus to use. The 'Bus CPLD' will control the generation of these signals and allow for parallel programming of the FPGA via its interface connected to the peripheral bus. FPGA devices have some internal RAM blocks that operate at very high speeds. For complicated processing algorithms however, more memory is required than available in the device (XCV200E has 114kbits). For this reason, a high-speed 4Mbit SRAM device will be provided with a dedicated interface to the FPGA.

The system requires a standard serial connection for use as a system console and debugging. Devices are available that will provide this interface which would interface to the peripheral bus. The logic used to implement RS-232 links for example on the other hand uses relatively few resources in an FPGA. By using the FPGA to implement the serial link, the flexibility to change the physical and software interfaces when required is provided. For these reasons, it was decided to implement the serial port functions in the FPGA.

The processor requires non-volatile memory for program storage if a node is to operate stand-alone. A Linux kernel and simple filesystem, which is enough to get the system fully functional requires at least 1-2MB of storage. For this reason, a 16Mbit (2MB) FLASH device will be provided for processor boot-up. The FPGA logic designs may also be required to reside in non-volatile storage. The XCV200E FPGA that was to be used requires 1.442Mbits (180kB) per configuration. The FPGA device used however is upward compatible to a 600,000 gate device which will require 4Mbits of configuration. In order to accommodate for this and to provide for the possibility of storing multiple configurations, an 8Mbit FLASH device was chosen. This will allow for up to five configurations for the XCV200E to be stored.

The number of LVDS channels provided will depend on the available pin resources

after the routing of other components to the FPGA and the number of lines available to off board connectors.

A watchdog timer and real-time clock device are desirable for system time keeping and recovering from system errors. These devices typically interface to a microprocessor bus or provide a serial interface such as I^2C. The choice of device will largely be decided by component availability and interfacing complexity required. The lower the complexity the more favourable the device.

If cost and PCB space allows, a user FLASH device may be provided for miscellaneous data storage.

Finally, the various components require power supplies that are capable of supplying the required current in addition to complying with device tolerances in terms of noise and voltage accuracy. The processor and LVDS I/O blocks of the FPGA require a 2.5V supply, the CPLD devices, GT64115 and FPGA I/O require 3.3V and the clock oscillator requires a 5V supply. Switch mode power supplies are most likely to be used due to the relatively high current requirements and to reduce power loss.

5.4 Functional Unit Design

This section gives a detailed description of the design process for each of the major functional units in the design.

5.4.1 Microprocessor

The chosen microprocessor was the IDT 79RC64574 64-bit MIPS processor [3]. The internal functional structure to the device is very simple and optimised for performance. As can be seen in figure 5.4, the processor has multiple control and execution units. The two execution units, the 64-bit Integer and Floating point unit operate in parallel, each with a five stage pipeline that is fed by a dual-issue instruction fetch unit. This allows both units to operate full speed simultainiously. The instruction fetch unit is coupled to the cache controller, which finds cached entries and performs external requests when cache misses occur. The processor also has a system control processor which manages interrupts, cache and memory management functions.

From the external view, the processor has clock, interrupt, configuration, JTAG and SysAD bus interfaces, which are shown in figure 5.5. The processor clock is derived from the system clock by multiplication in an on-chip Phase Locked Loop (PLL). The PLL is sensitive to noise and thus requires its own isolated power supply to ensure proper operation. This was provided by filtering the PLL power supply though an LC circuit, effectively acting as a low-pass filter. The processor has 7 external interrupt inputs: Six normal inputs and one Non-Maskable Interrupt (NMI). The sources of normal interrupts are: An interrupt from the memory controller that can be later decoded into memory errors, DMA transfers and PCI interrupts. An interrupt from the each of the FPGA, Bus CPLD and Config CPLD, which are to be used for application specific

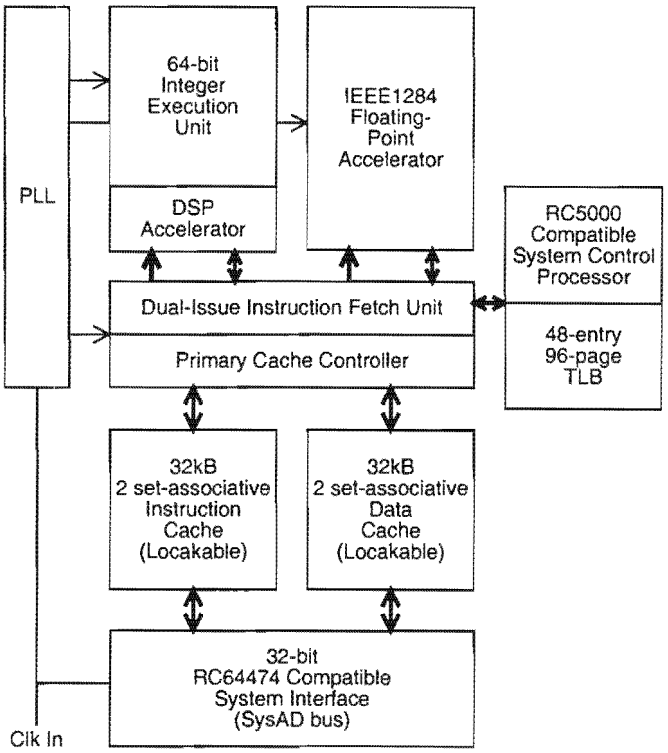


Figure 5.4: RC64574 Functional Block Diagram

purposes. The possibility for accepting interrupts from the PCI bus directly is allowed by routing a PCI interrupt line though the Bus CPLD, which has 5V tolerant inputs. The processor can only accept a maximum 3.3V input voltage.

The processor configuration interface has five signals that need monitoring and control. There are three reset signals which are generated from the Config CPLD and two signals used for processor configuration. The processor configuration interface uses a simple serial bit-stream. The Config CPLD can generate this and support for it is provided. In the event that this is not possible, provision is made for using a serial EEPROM device to configure the processor.

The processor supports the JTAG test and debug port and allows boundary scan and instruction execution. The JTAG interface can prove very useful for debugging and the processor was added to the system JTAG chain.

The processors 32-bit MIPS SysAD external interface matched the interface on the Galileo memory controller. A direct connection between the two devices along with the relevant pull-up and pull-down resistors required were added as per the recommendation provided in the device datasheets.

Other than these features, the processor provides no other on chip peripherals, which need to be interfaced.

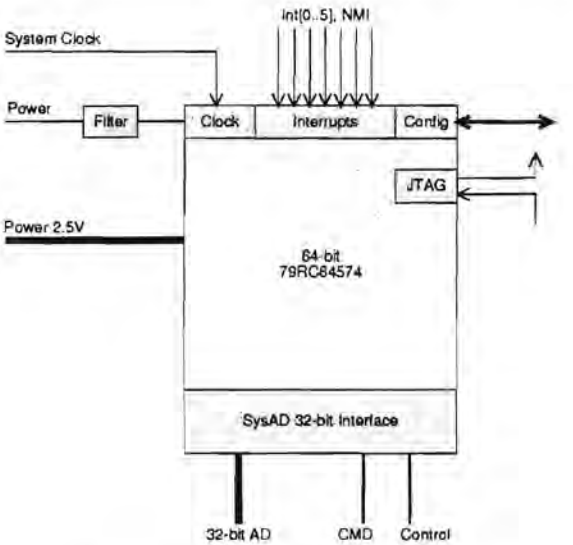


Figure 5.5: RC64574 Interfaces

5.4.2 Memory Controller

The Galileo GT64115 MIPS companion, PCI and memory controller device was selected to provide access to the PCI bus and memory devices. It supports three main interfaces, a 32-bit MIPS SysAD bus, a 32-bit PCI interface (up to 66MHz) and a 32-bit SDRAM and Device local bus interface. See figure 5.6. On the local bus, the memory controller supports up to four banks of SDRAM and provides five device chip-select signals, which includes a chip-select for the boot memory.

Like the processor, this device uses an internal PLL device for clock generation and required a power supply filter for correct operation.

The SysAD and PCI bus interfaces were connected up according to the specifications in the datasheets. For certain signals on the PCI bus more research and examples were needed to clarify their usage such as the PCI_PRSNT1/2 signals.

The local bus SDRAM interface was connected to two 16-bit SDRAM devices forming a single 32-bit 32MB SDRAM bank. The decision was made not to use an SDRAM module due to the difficulty in obtaining parts and the possibility of overloading and lengthening the high-speed bus. The FPGA and Bus CPLD devices are configurable logic devices and as such are very flexible. Because the chip-select signals are multiplexed on the local bus, all are available to each device. Thus, no specific chip-select signals are dedicated to either device. A convention was however specified for the usage of the chip-select lines, see Table 5.1. The peripheral bus bridge is the logic implemented in the Bus CPLD to provide transparent access to device on the peripheral bus from the local bus. The local bus chip-select signal will be decoded into address banks to address individual memory device.

The FPGA local bus interface was provided with the full set of signals required including the memory controller's multipurpose pins which can be configured to provide

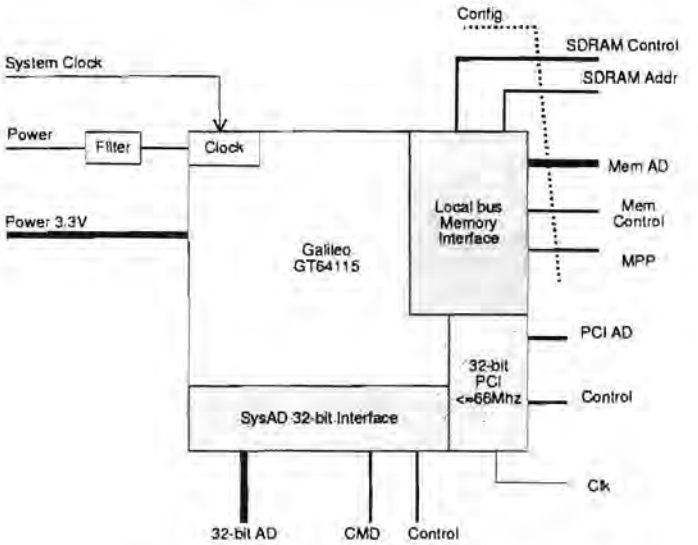


Figure 5.6: GT64115 Interfaces

Table 5.1: Memory device assignments

Chip-select signal	Assigned Device
SCS[0..1]	SDRAM Bank 0
SCS[2..3]	Not assigned
CS[0]	Peripheral Bus Bridge
CS[1]	FPGA
CS[2]	FPGA
CS[3]	Bus CPLD
Boot CS	Peripheral Bus Bridge

DMA support.

The Galileo memory controller device has a fully software configurable system memory map which allows reprogramming of device widths, timing and address ranges. For processor boot-up however a default configuration is need. The device provides two mechanisms for initial basic configuration. Firstly a series of week pull-up and pull-down resistors can be configured on certain local bus signals. On power up, the busses are tri-stated and the signal level on each of these lines is sampled. This allows various configuration options to be set. The alternative method is via auto-configuration whereby the memory controller reads a set of configuration values from a special location in the Boot memory device. Provision for both these modes was made.

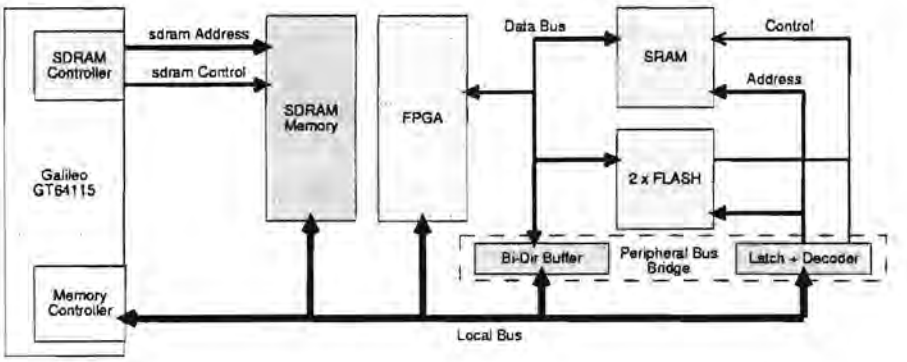


Figure 5.7: Local and Peripheral Bus Configuration

5.4.3 Bus CPLD and Peripheral Bus

The Bus CPLD device operation has been largely discussed in previous sections. The device chosen was a 144 macro-cell Xilinx CPLD in a 144 pin package [10]. This provided enough I/O pin resources to interface the local bus and peripheral bus along with providing various other interfaces. This device generates the address and control signals for the peripheral bus (master operation) and is a slave on the local bus. This device also generates the configuration signals required to program the FPGA. The system JTAG interface is routed to the I/O pins of the CPLD. This allows for the JTAG chain to be accessed from the CPLD and thus from the system its self with the correct logic design.

The peripheral bus was chosen to have an 8-bit (byte) wide data bus in order to reduce PCB routing complexity. The address bus was sized to address the maximum possible memory device on the bus. The processor boot FLASH is specified at 16Mbit but provision was made for supporting a 32Mbit device. This meant that the address bus needed to be 22 bits wide. $2^{22} words * 8bits = 32Mbits$. The SRAM device chosen for general-purpose storage and FPGA configuration cache was selected on the basis of availability. The device chosen was a 16-bit wide, 4Mbit device, which had already been purchased for other projects. This would not however directly interface to the 8-bit bus. Fortunately, the device selected has two control lines for byte wide accesses. During read and write cycles with only a single bank selected, the unselected bank is driven to tri-state thus not affecting that side of the bus. Therefore it was decided that both banks could be hardwired together making sure that each bank was selected individually. The Bus CPLD could then be programmed to use the two banks as separate chip-selects, effectively accessing the upper and lower banks of the SRAM as separate devices.

5.4.4 FPGA

The primary task for the FPGA in the requirements is hardware acceleration of processing algorithms. The FPGA however also needs to provide some essential system

support for the system to function. To interface to the processor, the FPGA is placed on the memory local bus to provide high speed 32-bit data access and provide the possibility for DMA transfers. The FPGA is required to implement a serial port controller (UART). This can be performed in logic, but a RS-232 level translator device is needed external to the FPGA to interface other RS-232 systems. An LT1386 EIA/TIA compatible line driver was used. It provided the capability for two UART channels however only a single pair of Transmit and Receive signals was routed. The reason for this was that there would not sufficient physical space on the board for the extra channel. The serial port will not be used as a primary communications interface in the final system and thus only one is required.

The FPGA is connected directly to a 16-bit high-speed 4Mbit SRAM device which will provide dedicated external storage space for the processing algorithms implemented. The JTAG interface of the FPGA is connected as part of the system JTAG chain.

To use the LVDS I/O capabilities of the FPGA, the I/O blocks with the LVDS channels to be used must be powered at 2.5V. Because all the other interfaces of the FPGA operate at 3.3V there was only a single I/O block available that could be used for LVDS. This I/O block provides support for eight LVDS pairs (configured four input and four output channels) and an LVDS clock input. The LVDS clock input was not implemented because the clock can be recovered with the aid of clock-data recovery techniques. CAT-5 cabling is suggested for LVDS signalling and thus two RJ-45 jacks were used providing four LVDS channels over two cables. This will allow for point-to-point communications between systems in a ring or square matrix configuration depending on the cable configuration.

Finally, the FPGA configuration interface was configured to allow parallel programming on the SelectMAP interface described in the datasheets and application notes.

5.4.5 Clocking

The processor, memory controller and SDRAM devices all require a synchronised in-phase clock to which they need to synchronise. The FPGA, Bus CPLD and Config CPLD also require clock inputs. The memory controller requires a clock greater than the frequency of the PCI clock (33MHz for PC environment) but no more than 75MHz. The processor requires a bus clock of between 33MHz and 125MHz and an internal pipeline clock greater than 100MHz. The SDRAM memory used has a maximum frequency of 100MHz. The CPLD devices each operate up to 100MHz and the FPGA is specified at around 200-300MHz.

The bus frequency was chosen to be 66MHz with the possibility of running up to 75MHz. With such a high frequency bus clock and the number of devices running from it, reliability problems can occur. If all the devices used the same clock, the loading on that signal would be too great and possibly cause increased jitter and signal degradation. The other problem is that each device will not be guaranteed to get

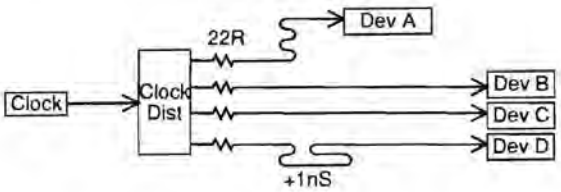


Figure 5.8: Clock Distribution Architecture

the clock in phase, which will cause synchronisation problems. To solve these problems, a clock distribution system needed to be designed with terminations to prevent reflections. A clock driver chip with ten output drivers was selected to manage the system clock. It offers a worst-case 350ps skew between output clocks. Because each device has its own dedicated clock signal, the nets are single ended. [11] indicates two recommended methods for terminating single ended clocks, source and end terminations. End terminations works well but require a resitor-capacitor network. Source termination requires only a single resistor at the signal source and was chosen for its simplicity. In addition to the low skew of the clock distribution device, the signal propagation delays on each net need to be very close to each other. For this reason, the net lengths on the PCB were equalised by routing in zigzag patterns were necessary to get the nets to within 2mm of each other.

One particular issue that was described in an errata document from the memory controller was that the clock signal to the particular processor (RC64574) needed to be slightly delayed from its own clock input. The simplest method of achieving this was to increase that clock net length to create the required signal delay of 1ns.

5.4.6 Power and Peripheral Devices

The power supplies for the system needed to be designed to be capable of handling the worst-case current requirements from the various devices. These worst case calculations are shown in table 5.2. The input voltage from the PCI is 5V and also 3.3V in newer systems. For the 3.3V and 1.8V supply, it was decided to use a buck mode (step-down) switching regulator from the 5V supply. This was mainly in order to reduce the heat dissipation that would result from a linear regulator. For the 2.5V supply, a linear regulator was used due to concerns about noise effects on the high-speed processor and LVDS channels using this supply. A heat sink was specified for use with this regulator. The switch mode supply needed special attention to be paid concerning the choice of inductors and smoothing capacitors. High ripple supply capacitors (Low ESR) and power inductors were used with values calculated from datasheet information.

A real-time clock with EEPROM memory, watchdog timer and voltage monitor device in an 8-pin SOIC package. This small device has a two wire I^2C bus interface connected to the FPGA that will implement a controller. This device satisfies the need for voltage monitor, watchdog and RTC in a single device.

Table 5.2: Worst case power supply calculations

Device	5V Supply	3.3V Supply	2.5V Supply	1.8V Supply
CPU RC64574 250MHz	-	-	880mA	-
Galileo GT64115	-	250mA	-	-
Xilinx XCV200E	-	400mA	20mA	1000mA
SDRAM (128Mbit)	-	2x140mA	-	-
SRAM (4Mbit) - 12nS	-	2x120mA	-	-
FLASH - 90ns	-	2x12mA	-	-
Xilinx XC9572XL	-	60mA	-	-
Xilinx XC96144XL	-	110mA	-	-
Clock Distribution	-	60mA	-	-
RTC XC1227A	-	2mA	-	-
LTC1386	-	400uA	-	-
66MHz clock	20mA	-	-	-
Total	20mA	1426mA	900mA	1000mA

5.5 System Configuration Options

One of the aims of the hardware design was to make the system flexible and configurable in order to provide a wide range of possible configurations and reduce the risk of a design problem with an inflexible design.

The configuration options provided for the Galileo memory controller are shown in Table 5.3. The interrupt and miscellaneous configuration is shown in Table 5.4.

5.6 Printed Circuit Board Design

The printed circuit board design had the constraint that the hardware needs be compatible with PCI in terms of physical size and layout. The board was specified to be a six layer design in order to keep manufacture prices lower although for the prototype, additional layers could be accepted if necessary. Components were restricted to single side with only very low profile passive components allowed on the back of the board as per the PCI clearance constraints. Where at all possible, surface mount components were used. The design was specified to keep EMC consideration in mind as the hardware was required to interface a PC system, which typically produces a lot of interference and may be susceptable as well. (See [12] for EMC design)

The most important aspects of the PCB design were the bus and high-speed track layout and the component placement to optimise this. The local bus is a 32-bit wide bus with additional control signals running at 66MHz to 75MHz. A circuit board trace typically requires termination when the propagation delay on the net exceeds the time

Table 5.3: Galileo GT64115 Configuration

Resistor/Jumper	Configuration
R43, R44	Swapped CS[3] and Boot CS (PCI BAR)
R45, R46	Swapped SCS[3:2] PCI BAR
R47, R48	PCI Expansion ROM enable
R49, R50	CS[3] and BootCS PCI enable
R51, R52	Internal Registers PCI enable
R53, R54	Autoload enable
R55, R56	CS[2:0] PCI enable
R57, R58	SCS[3:2] PCI enable
R61-R64	BootCS , CS[3] bus width
R65, R66	PCI Conditional Retry
R35-R42	PLL Configuration

Table 5.4: Interrupt and other Configuration

Resistor/Jumper	Configuration
J5	GT64115 Int Enable
J6	PCI Int B to CPLD
J7	PCI Power Required (Open 15W, Closed 7.5W)
J8	PCI Clock to System Clock
J10	FPGA - No Config Pullups
J11	FPGA - Config Pullups
DIP SW 1..0	CPU PLL: '0' - x2, '1' - X3, '2' - x4, '3' - x5
DIP SW 2	CPU Timer Int 5 Enable
DIP SW 4..3	CPU Write Mode: '0' - R4400, '1' - Res., '2' - Pipeline, '3' - Reissue
DIP SW 5	ADDR to Data Delay, '0' - Slow, '1' - Fast

of 1/10th of a wavelength. Therefore:

$$\begin{aligned} 300000kms^{-1} &= t = 300000000ms^{-1} \\ t/75MHz &= 4m(\lambda) \\ \frac{1}{10}\lambda &= 40cm \\ FR4 &- X\%\,vel \\ max\,net\,length &= 20cm \end{aligned}$$

It was therefore important to keep the bus net lengths normalised and shorter than 20cm in order to avoid having to terminate the bus.

Due to the specification to try stick to a six layer board, a lot of effort was required achieve this. The hardware design uses four separate supply voltages. Fortunately, the supplies are each only required in localised areas on the board. This allowed the use of segmented power plains, reducing the need for additional physical plains.

All the clock and bus nets were routed first in an effort to minimise the number of layer changes and net lengths. Design rules such as no 90o corners and equal track spacing and widths were used to help improve signal condition. All bus nets were routed in a daisy-balanced configuration and the bus devices were arranged to support this configuration. The support and low speed signals were routed around the critical nets towards the end of the PCB design phase once all critical paths had been finalised.

Various test points for the power supplies were placed around the board for testing the board. Some of the remaining free I/O lines on the FPGA were routed to a DEBUG header for testing purposes.

The switch mode power supplies and smoothing capacitors were placed together and as far as possible from the logic devices. This was an attempt to keep the device power supply voltages as noise free as possible. Each device power pin was additionally decoupled with a 100nF capacitor for high frequency supply demand changes and 22uF tantalum capacitors were distributed around the board to help with lower frequency supply demand changes.

The LVDS and DB9 serial port connectors were placed on the East edge of the board so that they would be accessible from the PCI slot interface panel of a standard PC. The PCI edge connector was hard Gold plated for improved contact conductance. Small copper planes were placed underneath the power supply devices with Vias though the board to help increase the surface area for heat dissipation. A ground plane was placed beneath the processor device on the top layer as the processor contains an exposed metal heat sink on its underside.

The entire design was hand-routed due to the complicated power plane design and special high-speed routing requirements. This allowed precise control of the board design to meet the requirements. The SDRAM devices were routed in a manner that optimised the net placement however it required track sizes smaller than the manufacturers recommended minimum track widths. After consultation with the manufacturer, it was confirmed that localised locations on the top and bottom layers may have track

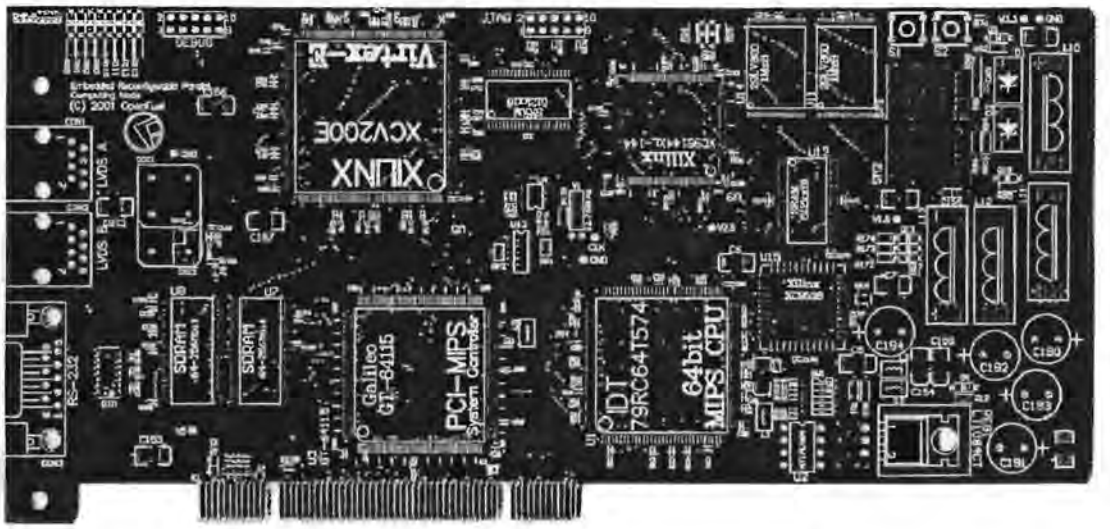


Figure 5.9: Top side, unpopulated project hardware

widths smaller than the minimum size specified. This greatly simplified the SDRAM routing which otherwise may not have worked on the six layer constraint.

5.7 Conclusion

This chapter gave a quick summary of the hardware design process and some of the decisions made. The result of this work is a hardware design capable of meeting the system requirements. Each node has a microprocessor with boot memory and high-density program memory, an FPGA for system implementing system interfaces and processing and high speed communications links for inter-node communications.

Four PCBs were manufactured at Trax Interconnect and the majority of the components were placed by Rhomco electronic assembly services.

This hardware design is a first revision prototype design and thus was designed with a major focus on risk reduction. This meant that the fastest and most optimal solutions were not always favoured over more reliable and proven ones. Future projects may enhance the design to improve performance and features.

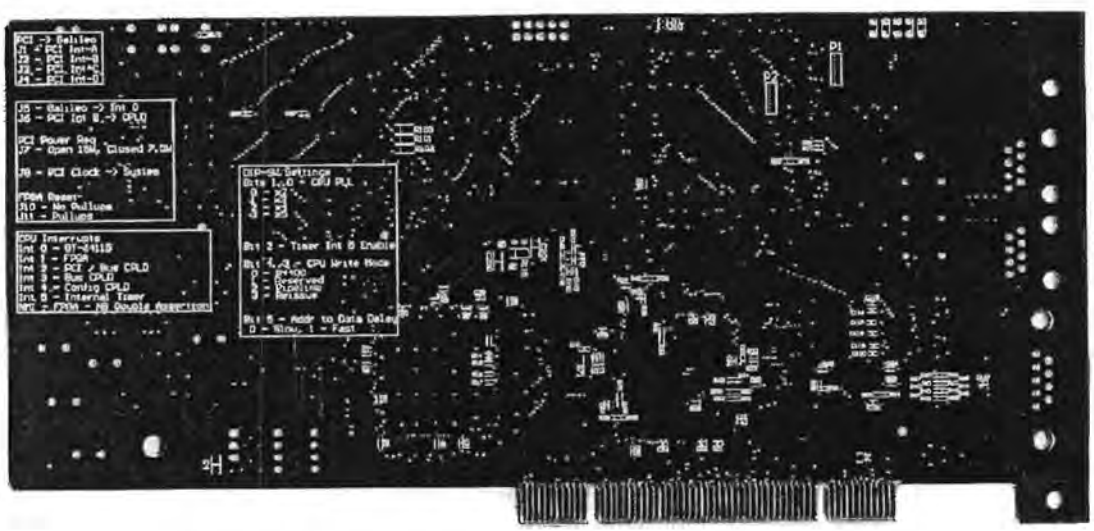


Figure 5.10: Bottom side, unpopulated project hardware

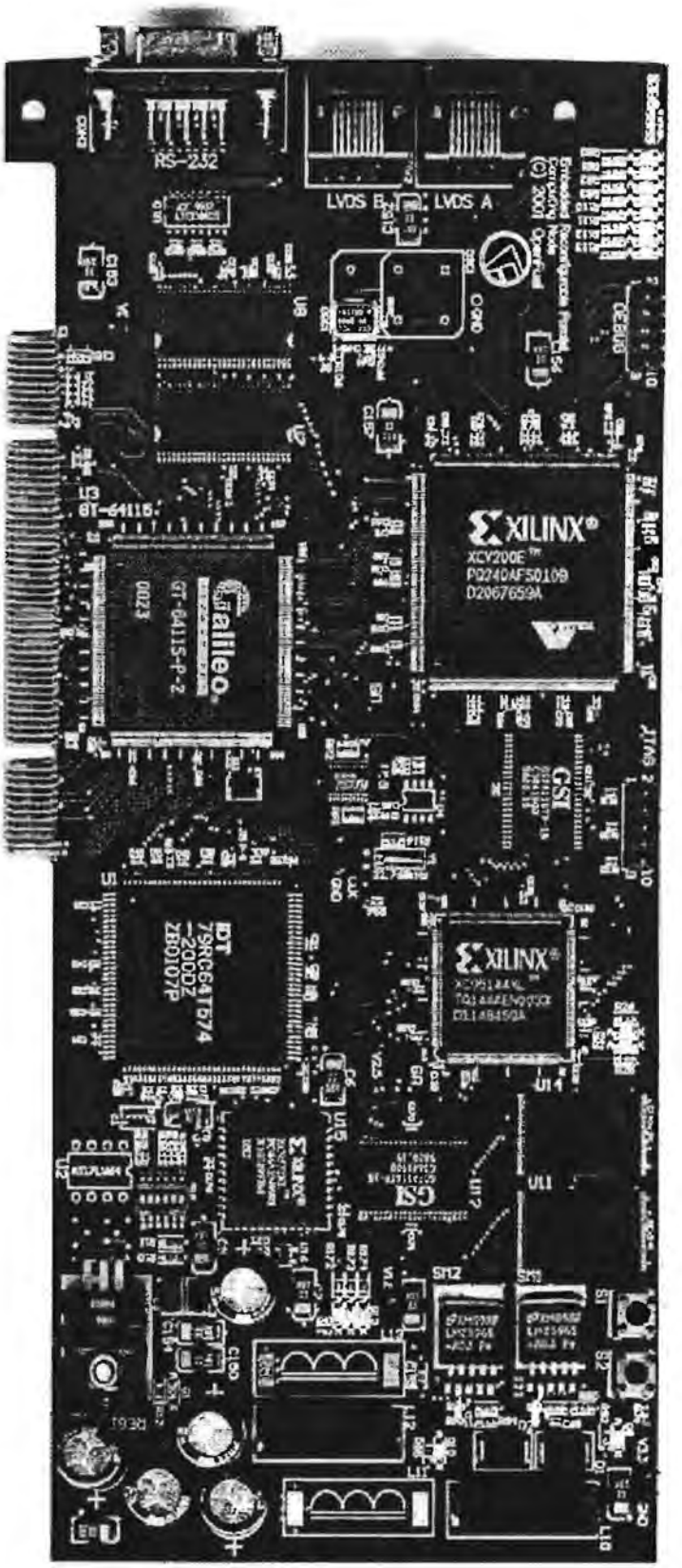


Figure 5.11: Top side view, populated project hardware

Chapter 6

Hardware Verification

This chapter describes the process of verifying the design and testing the hardware for errors and compliance. This process was on going and ran in parallel with the firmware and software design phases describes in the following two chapters.

This chapter begins with a description of the initial testing performed on the PCBs before the placing any of the components. This is followed by a description of the testing of the power supplies and the configurable logic devices. The testing of the memory controller was performed and testing the PCI interface. Finally when the whole system was functioning the processor operation was tested and verified.

6.1 PCB Inspection

The PCB inspection was relatively simple. Each individual PCB had already been flying probe tested after manufacture to the compliance of the Gerber design information provided. However, errors in the design may have existed which would not have been detected. The most important feature to test was the power plane connections and isolation. If any short-circuits were present between any of the power planes, the hardware will not function and could be damaged if powered up.

A simple digital multimeter was used to test the power plane isolation as well as testing for continuity between distant points of the same supply plane. All the planes were verified to be isolated from each other as required and the planes were all continuous from source to load.

A quick visual inspection of the design was also performed with a printout of the PCB design as a reference. Everything appeared to be correctly manufactured.

6.2 Power Supply

The switch mode power supply devices selected for the project had not been previously used or tested and thus a full compliance testing was performed to verify their operation. The power supply devices and all the supporting components were manually

placed on the PCB in the laboratory. For each supply, the power was slowly applied to the board by increasing the current limit until constant voltage operation was achieved. If any problems existed, the current consumption would greatly exceed the power supplies quiescent current requirements. By using the current limit, damage to the devices can be prevented.

The switch mode power supplies were initially verified to generate the correct output voltage levels at 'no load' conditions even though the device specifications indicated that a load was required. Resistive loads were used to progressively test the power supply output current up to the rated 3A. The output voltage was observed to drop slightly with the increasing load but never fell below the specifications. An oscilloscope was used to measure the ripple voltage produced. Initially, the ripple voltage was measured at 80mV(rms) however after placing the smoothing capacitors this was reduced to levels below the noise level of the oscilloscope (<2mV). Device power supply ripple and noise tolerances are not given for most of the devices however, noise levels in a standard PC environment are generally orders of magnitude greater than this and thus correct device operation should be assumed. During the 3A load test, the heat dissipation observed from the switch mode devices was negligible, only becoming slightly warm to the touch ($\pm 40°C$).

The 2.5V supply is generated from the linear regulator and was also tested. No detectable noise was observed on the output. Linear regulator devices are typically very stable and with low noise and do not normally cause problems. The only problem with them is power dissipation. The heat generated by the device with a 1A load from 5 Volts is 2.5W. With the heatsink rating of $57°C$ @ 2W and additional PCB designed heat removal, no overheating was observed.

The power supply devices were found to produce the correct output voltages with no detectable noise and each was capable of supplying the specified maximum current requirements. Finally, the clock and clock distribution components were placed and their operation verified. The clocks are critical for the functioning of the logic devices in the system. The 1ns clock delay designed for the processor was difficult to verify due to the sampling frequency of the oscilloscope available.

6.3 Configurable Logic

The passive devices (resistors and capacitors), memory, clock and the configurable logic devices were then placed by a contract manufacturer. The first tests to be conducted were to again test the power supply isolation. Any design errors in which power pins had been routed to ground pins for instance would immediately be apparent. The tests showed that at least no power pins and ground pins had been wrongly connected.

Power was applied to the board by slowly increasing the current limit until the power supply switched into voltage limit mode meaning that no short circuits or excessive current drawing problems were present. If the current usage of the board had exceeded a limit specified by totalling the quiescent current requirements of each de-

vice, a fault could be assumed. Further testing would have been necessary to find the fault.

It was verified that power on the board was not causing any of the devices to heat-up and current consumption was as expected. Verification of the configurable logic devices was started. All the configurable logic devices used have a JTAG test and access port. The primary objective was to establish communications with the devices using the JTAG interface in order to configure them.

A link was used to bridge the processor JTAG chain link as the device had not yet been placed on the PCB. A Xilinx JTAG downloader to interface a PC to the hardware was built on VeroBoard as per the Xilinx provided design.

Initial attempts to probe the hardware using the JTAG port failed. The software indicated that no devices were present in the chain. After some investigation, it was found that the problem was with the JTAG download hardware and a jumper for the FPGA configuration mode needed to be set. The JTAG software finally detected the correct devices in the right order on the JTAG chain. This verified that the power supplies to each device were correct and each device was operating internally.

The FPGA was the first device to be tested. A simple design to flash the LEDs was written in VHDL. The software compiled the code correctly however the configuration via the JTAG interface although indicating correct operation, produced no change in the state of the FPGA. A setting in the software configuration options was found to cause this problem and the FPGA configuration was successfully tested. Similar tests were used to check the functionality of the Bus CPLD and Config CPLD.

The Config CPLD VHLD design was the first to be tested. While no devices depending on the state of the reset signals were present, the reset timing delays were tested and measured on the oscilloscope. They were verified to be the same as those calculated for the design. The push-button inputs were tested and verified to generate the correct behaviour and ensure that the debouncing logic was functional. The brown-out reset detection of the MAX811 device was also verified by reducing the power supply current limit to the point that the system supply voltage started to drop. This triggered the MAX811 to assert its reset output and caused the CPLD to perform the programmed response.

Work then began on the emulation of the local-bus to test the Bus CPLD design, this is described in section 9.1.2.

6.4 Memory Controller

Once the correct operation of the configurable logic devices and peripheral bus memory was established, the processor and memory controller devices were placed on the PCB. The Config CPLD was programmed to keep the processor in reset during this testing phase. Again, a test of the power planes was conducted to check for PCB design errors. These tests found no problems with the memory controller PCB design. Resistors to specify the power-up configuration of the GT64115 were then soldered

into position. The device was setup for a little-Endian default configuration.

On providing power to the system, it was observed that the memory controller device was warm to the touch. Information about the power usage of the device was not provided. The datasheet recommended the use of a heatsink when airflow was reduced. This removed any concerns about the devices operating temperature.

The first major design test that was performed was interfacing the node hardware to a PC by inserting it into a PCI slot of the host motherboard. Before this test, every pin on the PCI edge connector was double checked to make sure that the correct supply nets and device pins were connected to the interface. Any problems could cause a critical failure of the processor card or PC hardware because the PC power supply is capable of providing enough current to 'burn' a short-circuit, damaging the hardware. Once it was decided that the hardware was ready for testing in the PC environment, an unused Pentium 133MHz computer was used as the host.

On the first power up with the hardware in the PC, the PC system did not respond and start booting up. This caused concern and the power was turned off. After an inspection it was found that nothing on the card had heated up or seemed unusual. The entire design was rechecked and during the process, a feature called 'PCI Retry Enable' which is a configuration option of the GT64115 was identified. The initial interpretation of the use of this configuration option was incorrect. This feature was enabled which meant that the memory controller forced PCI retry operations until it received a signal from its local processor to indicate that PCI transactions may proceed. The CPU was in reset with no code present for it to run. The memory controller thus kept the entire PC system waiting indefinitely. By simply changing the resistor setting, the PC system started up correctly with the hardware and identified the Galileo memory controller device on the PCI bus. The Linux operating system on the host's PCI probe identified the device but no drivers were available as expected for the device. This all verified the correct layout and design of the PCI interface on the Galileo memory controller device.

The next phase of testing the memory controller was to access the memory on the processor card from the PC across the PCI bus. This was very important in that it would verify the operation of the entire memory controller before work on getting the processor working started. The memory controller interprets PCI requests in the same ways as processor requests, thus if PCI accesses worked then it could be assumed that the memory bus was operational.

A driver for the host PCI PC computer running Linux was written. This was firstly used to test access to the SRAM device on the Peripheral Bus. Various problems were overcome and a greater knowledge of the memory controller and PCI systems was gained. After finding and fixing various issues, the PC was able to read from the SRAM device through the Config CPLD and memory controller from the PC.

The SDRAM was the last component of the memory system to be verified. A special intialization sequence is required to by the SDRAM devices in order to initialise the interface. Some support for the special bus commands sent to the SDRAM are provided by the memory controller device. The SDRAM initialisation sequence was

sent from the PC Linux driver and the SDRAM was tested. The SDRAM was setup for 2CLK precharge, 4 bank interleaving, 2CLK CAS/RAS latency and 8 data bursts. Two prototype boards were being built during this hardware verification stage. On one board, the SDRAM was working while on the other, specific bits in the data read-back from the devices were incorrect. It was found that some of the SDRAM pins on the one card had 'dry joints' or pins that had not soldered correctly. This was caused partly because the PCB footprint for the SDRAM devices was too small and solder did not make proper contact on all the pins. Each dry-joint needed to be carefully re-soldered and tested. This solved the problem with the SDRAM memory.

6.5 Processor

The processor was placed at the same time as the memory controller although it was initially held in reset. During the PCB testing phase, a very low-impedance was detected between the power and ground of the processor. After closer inspection it was discovered that a design error had caused the processors PLL power supply pins to be connected the wrong way around. Fortunately the PLL supply was filtered and thus moving the components and using link wire corrected the problem. On one of the boards, a low-impedance still existed between the processor power supplies in the order of 45Ω. The cause of this low impedance was never found however device operation was not affected.

The processor operation was only tested once the memory system was verified to be operational. The first step taken was to test the CPU 'ColdReset' serial configuration sequence. This involved testing the Config CPLD code and using an oscilloscope on the serial channel to capture and verify the sequence. The processor PLL supply was also tested with the oscilloscope to check for supply ripple characteristics from the PLL. A week 200MHz ripple verified that the on chip PLL was operating in X3 mode as specified: $66.7MHz * 3 = 200MHz$. Various changes to the CPLD code were implemented in order to generate the correct configuration sequences.

The processor was then ready to be booted. The memory controller was configured to allow the CPU to boot from the SDRAM memory. A test program was loaded into the SDRAM and the processor reset released. An oscilloscope was used to monitor the processor's SysAD Bus interface to the memory controller to check for bus activity. After various processor configuration changes, the system setup was correct and the processor operation was verified by running test code and verifying the results across the PCI bus.

Chapter 7

Firmware Implementation

The ERPCN01 node hardware contains three configurable logic devices in which firmware logic designs are required to be implemented in order for the hardware to function. The design language used for the majority of designs implemented as part of this thesis was VHDL. This language provides a method for describing digital logic designs in a high level programming language style. Some of the important VHDL design implications are discussed in the first section of this chapter.

Each individual design used during the course of the project, including test and verification designs as well as final firmware designs is described. Each design is documented with a requirements statement, specification, various design options and choices made as well as implementation details.

7.1 VHDL design implications

VHDL stands for VHSIC Hardware Description Language, VHSIC meaning Very High Speed Integrated Circuit. It is an IEEE standard for logic description which resembles a computer programming language.

Designing and implementing logic designs requires very specialised software that is generally manufacture specific. The design process which is generally followed is as follows: A design is started from a set of requirements, which are translated into a specification. VHDL code is written to describe the design in a standard programming language style text file format. Very sophisticated and complex compiler software is then used to infer logic-elements and connections from the VHDL descriptions. A net-list is generated as the result of this operation that a synthesiser will accept. This net-list is usually not hardware dependent but may contain elements not directly translatable to the target device technology. This net-list and various other net-lists from other sources such as vendor specific macros are then merged together in the synthesiser that generates a design consisting only of elements available in the target logic devices architecture. Place-and-Route software is then employed to allocate logic and routing resources in target device to implement the design. In FPGA devices, the rout-

ing delay between logic elements is path and location specific. Thus the place and route process is comprised of an iterative placement and net-delay analysis procedure that attempts to optimise the performance the design to a set of user specified design constraints. The logic designer needs to be very aware of the capabilities of the compiler and target technology used. Very complicated logic equations or circuits can take a long time to produce a valid result. This time may be longer than the period of the clock used for the design. This then requires a redesign of that block to either reduce the complexity or use techniques such and pipelining and manual placement to achieve the performance goals. The designer also needs to be aware of the various VHDL constructs and what logic elements are instantiated or inferred from them. Often, the incorrect usage of language constructs can create a design that either performs very poorly or not at all, even when the VHDL appears to be logically correct.

At two stages during the design process, the entire design or a particular module may be simulated. The logical operation of the system may be simulated directly from the compiled net-list. This provides no timing information but can be used to quickly test the design logic. This simulation does not require a full compile process and contains no routing and timing information and is thus much faster in runtime. The final design as implemented on the device may be simulated which includes actual silicon path and logic element delay information. This type of simulation can be very useful to analyse asynchronous operations and the response to external stimulus or the devices own output.

All the CPLD and FPGA devices used in the ERPCN01 hardware are Xilinx products and use the same compiler and synthesis tools. The compiler used in this project was a Xilinx vendor release of the Synopsis FPGA Express compiler, version 3.5.1. The Xilinx Foundation 3.1-r8 software package was used for all the logic design and simulation.

7.2 Configuration CPLD

The configuration CPLD device is responsible for system reset control and processor boot-up configuration. It also has non-dedicated connections to the Bus CPLD and FPGA for general-purpose usage and interrupt line to the processor is also provided but unassigned.

System reset control is the primary function of this CPLD. The project hardware has three system resets and five reset sources capable of producing different reset conditions and sequences. The Bus CPLD device is responsible for monitoring the reset inputs and synchronising the reset outputs. It also needs to generate the timings required by the various devices it controls.

Only a single design was implemented on this device due to its specific function. The design under went a series of developments that fixed problems and added or removed features as required.

7.2.1 Requirements

MIPS Microprocessor Requirements

The IDT 79RC64574 microprocessor device has three reset inputs and two configuration lines that need to be interfaced and controlled in a specific manner for the proper operation of the device.

The three reset inputs are: 'VCCOK' - indicates the state of the system supply voltage and acts as a global reset. 'ColdReset' - used at power on and during normal operation to initiate a 'Cold Reset' that requires the same sequence as a power-on reset. 'Reset' - is used at power on and during normal operation to perform a 'Warm Reset' after the device has been initialised from a 'Cold Reset'. After a 'Warm Reset' the processor starts executing system code from the boot-vector.

The datasheet for the '574 specifies the timing diagrams for boot-time initiation and reset, shown in Appendix X. The requirements laid out by the processor manual [x] (chapter 12) and taken from the timing diagrams are:

- The interface must be synchronous with the processor/CPU clock input and comply with the timing requirement of the device.
- 'VCCOK' must be asserted for a minimum of 100ms after the supply voltage has stabilised in the processors operational voltage range.
- 'ColdReset' must be asserted at least 64k clock cycles after the assertion of 'VCCOK' and must be de-asserted synchronously with the CPU clock.
- 'Reset' must be asserted until at least 64 clock cycles after 'ColdReset' has been de-asserted. It must be de-asserted synchronously with the CPU clock.
- At least 256 CPU clock cycles after the assertion of 'VCCOK', the initialisation interface generates a clock on the 'MODECLK' output and samples a 256 bit initialisation bit-stream synchronised with the 'MODECLK', which must be provided.
- The 'MODECLK' has a period of 256 CPU clock cycles and the device suppling the bit-stream must provide a stable output before each rising edge of this clock.

The format of the processor initialisation bit-stream is shown in Table X in Appendix X. This bit-stream is required to be generated by the Config CPLD or the Serial EEPROM device.

System Requirements

The devices other than the processor that are controlled or affected by the Config CPLD are the FPGA, Bus Control CPLD, Reset Voltage Monitor, Serial EEPROM device and Galileo MIPS companion chip. Two push button inputs are also available to the Config CPLD, which are required to be de-bounced in logic. Their requirements are:

- The Config CPLD and FPGA require a reset input signal that is de-asserted after the supply voltage has stabilised and before the processor is released from reset.
- The Reset Monitor Device has a reset input for manual reset generation, which needs to be asserted to emulate a system cold reset.
- The serial EEPROM has a chip enable and output enable inputs which need to be generated when that device is used for processor initialisation.
- The 'mTest' input of the Galileo device is routed to the Config CPLD and needs to be asserted for the device to enter normal operation.

7.2.2 Specification and Design

The design for the Config CPLD will be synchronous with the 'ConfCLK' input clock which is synchronous with clocks to the processor, bus CPLD, companion chip, SDRAM and FPGA. This will allow output signals to the processor to meet its synchronisation requirements. All reset input sources are synchronised with the system clock.

The input clock is nominally 66.667MHz to a maximum of 75MHz. A 23-bit pre-scaler counter will be used to generate the timing requirements. This will give a worst case minimum count period of $2^{23}/(75 * 10^6) = 0.112s = 112ms$. This counter is started after the de-assertion of all reset inputs.

The 'SystemReset' signal controlling the FPGA and Bus CPLD is asserted until half the pre-scaler count value is reached after a reset input event: $2^{22}/(75 * 10^6) = 0.559s \approx 56ms$.

The 'VCCOK' signal is de-asserted after the first pre-scaler roll-over (>=112ms).

The completion of the bit-stream initialisation signal is asserted at the first pre-scaler count value with bits 15 and 16 set following the assertion of 'VCCOK'. This effectively allows for $2^{16} + 2^{15} = 98304clocks$ in which to perform the initialisation, which needs at least $256bits * 256clocks = 65536clocks$ for completion.

The initialisation sequence is reset by the de-assertion of the 'VCCOK' signal and is clocked by the 'MODECLK' input from the processor. Certain bits in the stream are fixed and the rest are sampled from the inputs of an 8-bit dip-switch, which allows various bits in the stream to be modified without changing the VHDL design and re-programming the CPLD device.

The 'ColdReset' signal is de-asserted at the next 8-bit overflow of the pre-scaler following the de-assertion of 'VCCOK' and the completion of the bit-stream initialisation.

The CPU's 'Reset' input is controlled by the boot-up sequence but is also required to be toggled without a full system reset to enable 'Soft Resets'. This signal is asserted by either a system or soft reset input signal and can only be de-asserted when 'ColdReset' is de-asserted. A pair of delay registers is triggered following the de-assertion

of the controlling signals which keeps the CPU 'Reset' signal delayed at least 64 clock cycles after the de-assertion of 'ColdReset' as required.

The two push-button inputs are used as 'SystemReset' and 'Cold Reset' inputs. The 'SystemReset' push-button is de-bounced and the 'Cold Reset' push-button signal is routed to the external MAX811 Voltage Reset Monitor device. The input from the MAX811 is used to generate the Config CPLD's 'Cold Reset' events.

Finally, three LED devices are connected to the Config CPLD. These were used to indicate the status of the 'ColdReset', CPU 'Reset' and 'SystemReset' signals.

7.2.3 Implementation

The VHDL design was written according to the specifications and simulated test the designs compliance to the requirements. During the course of the implementation process, minor errors in the implementation and specification were discovered from the simulation results and corrected.

The Config CPLD was originally specified in the hardware design to be a 36 macro-cell device, however a pin compatible 72-macro cell pin-compatible device was used. The 36 macro cell device would not have had the required resources for this design. The results of the final design resource utilisation after implementation are:

75% of macro cells, 30% of product terms, 65% of registers and 46% of function block inputs used.
'ConfCLK' maximum input frequency 100MHz, 'ModeCLK' maximum input frequency 71MHz

7.3 Bus + Control CPLD

The Bus and Control CPLD has three main functions. Most importantly, it implements a bridge between the system local bus and the peripheral bus allowing transparent bus transactions to be performed across it. Secondly, it has the task of handling FPGA run-time reconfiguration. Lastly, it provides a memory mapped system control interface for processor reset control and FPGA configuration initialisation.

7.3.1 Requirements

Local to Peripheral Bus Bridge

The local to peripheral bus bridge needs to interface the standard 8-bit FLASH and SRAM devices with separate address and data busses to the Galileo memory controller's 32-bit multiplexed Address/Data bus.

The timing requirements on the bus are very demanding and a proper understanding of the bus operation was required in order to specify and implement the design. The memory local bus is multiplexed meaning that the address along with control lines are

shared with the data bus connections. Various signals are provided to latch and qualify these signals which, need to be decoded. The bus also supports up to 8-word burst read/write transfers to a selected device. This is possible due to the fact that the lower three address bits are provided as dedicated pins on the Galileo device. A detailed description of the memory controller's memory bus (local bus) along with full timing diagrams are located in Galileo GT64115 datasheet [Chapter 5 Section 8].

The CPLD is required to bridge single and multi-word read/write cycles from the local bus to peripheral bus. The CPLD is also required to perform address decoding to fragment the local bus chip-select 0 (CS0) and 'BootCS' windows into smaller memory windows to each device on the peripheral bus. The memory controller's 'BootCS' signal must access the processor boot memory device.

The peripheral bus is has an 8-bit data bus with a 22-bit address bus and operates as a standard asynchronous bus. The CPLD is required to generate appropriate chip-select lines to the individual devices and generate the read/write signals.

The CPLD is finally also provided with an interrupt input from the PCI bus and has two interrupt lines to the processor. These lines must be de-asserted if not in use. Finally, a connection the the systems JTAG chain is provided. This is for future work and it is required to be driven tri-state.

FPGA Configuration

The Bus + Control CPLD is required to generate the signals needed to configure the Xilinx Virtex-E FPGA device via the SelectMAP [x] interface. This basically involves generating the various control signals to initiate and control the configuration and generating an address and control signals for the memory device on the peripheral bus that holds the FPGA design bit-codes.

During this configuration mode, the peripheral bus is busy and may not be accessed from the local bus. All requests to the peripheral bus should be ignored.

The design must allow for power-up as well as run-time configuration and reconfiguration. A maximum of configuration speed of 50MHz may be used without the need for using a more complicated configuration algorithm. A detailed description of the Virtex-E configuration via the SelectMAP interface is provided in the Xilinx Application Notes [15] [16]

System Control Interface

The system control interface is required to provide an interface to allow the state of the system to be controlled without physical intervention. It is required to provide an interface that will allow the processor reset to be asserted, de-asserted and pulsed (off->on). This will allow the state of the processor to be controlled either by its self or from across the PCI bus from the host. It is also required to provide an interface to initiate FPGA configuration. No requirement was set for the selection of an FPGA configuration memory device or address. Finally, an interface to allow the toggling

of the 'SystemReset' signal and a method to acquire the systems revision number for software purposes.

7.3.2 Specification and Design

The specification and design process for this component of the system went though a series of revisions before reaching its final state. During various hardware test and verification processes, slight modifications and changes in functionality were required and some features were only implemented when required.

The design is required to operate in two distinct modes: FPGA configuration and bus bridge modes. A global signal will indicate this state and disable the appropriate logic and act as the control signal for bus multiplexers. Of the signals that need to be multiplexed, the address and data on the peripheral bus have two possible sources, one from the FPGA configuration logic and the other decoded from the local bus. The peripheral bus control signals need to be generated for both modes.

The FPGA configuration requires a configuration clock. 8-bit parallel data is latched on every rising edge of this clock. The FPGA interface is signalled that a new configuration cycle should begin. A read request starting from address zero is directed to the memory device containing the configuration data from the Bus CPLD. The memory device outputs this data onto the bus which is connected to the FPGA interface. During each configuration clock cycle, the memory address is incremented thus loading the entire design into the FPGA. On completion, the FPGA notifies the Bus CPLD with a 'DONE', which returns the CPLD back to the bridge mode of operation.

The local-bus interface upon closer inspection is not very complicated, but requires precise timing. The address, chip-selects and control signals need to be latched on the falling edge of the 'ALE' (Address Latch Enable) signal from the memory controller. A chip-select timing signal from the memory controller is used to qualify the latched control signals. The peripheral bus read and write signals are generated directly from the decoded signals from the local-bus. Bi-directional buffers are controlled by the decoded read/write signals to allow the data bus to be bridged to the local-bus during data transfers.

The system control interface is to be implemented as a memory mapped virtual address on an unused chip-select signal. The communication lines between the Bus CPLD and Config CPLD are to be used to manipulate the Reset signals. Any access to the virtual device (chip select) will invoke an action selected by the address selected. A table of 'commands' is provided in Table 7.1.

7.3.3 Implementation

The design was written as a single VHDL file and targeted to the 144 macro-cell Bus CPLD. The Bus CPLD design was tested initially in simulations to verify correct operation. During the simulations, errors in the design and implementation were found

Table 7.1: System control interface, on Chip Select 3 (CS3)

Address	Command	Action
0x000	Reset	Asserts the processor Reset signal
0x010	Run	Deasserts the processor Reset signal
0x020	Pulse Reset	Asserts then deasserts the processor Reset
0x080	Toggle Sys Reset	Changes the state of the system reset
0x100	Revision	Places System Revision number on the bus

and corrected until the simulated operation conformed to the requirements of the local bus interface and other requirements.

The initial testing of the design in the system was performed during the hardware verification phase described in chapter 6. The FPGA was used initially to emulate the local bus while the memory controller device was not present in the system. A more detailed description of that process is described in the FPGA Designs, section 7.4.6.

The design did not give many problems although during the hardware verification, a problem caused by a dry joint on the FPGA Flash, which prevented the SRAM from functioning correctly. This prompted an investigation that looked partly at the CPLD design before the cause was found.

The final design implemented in the Bus CPLD required the following resources: 95% of macro cells used, 35% of product terms used, 38% of registers used and 50% of functional block inputs used.
The maximum clock frequency of the design was limited to 100MHz.

7.4 FPGA Designs

During the development of the project, 10 basic designs were developed to test and provide interfaces for the system. Some designs underwent constant review, additions and modifications while others were very simple tests. The simple designs are described briefly with more attention being paid to the complicated designs.

All the FPGA designs interfacing the system local bus are specified to use the WishBone™System on Chip (SOC) bus. This is an open standard for linking modules of a system together with a common interface. The design requirements and specifications for implementing Wishbone compatible systems were analysed and followed in all the designs. By sticking strictly to a single interconnect system, design reuse was greatly improved.

7.4.1 Basic LED Test

This design served as a simple example in order to verify the hardware design and the software tools used to implement the FPGA designs. Initially, this design consisted only of a counter dividing the system clock down to a human detectable frequency

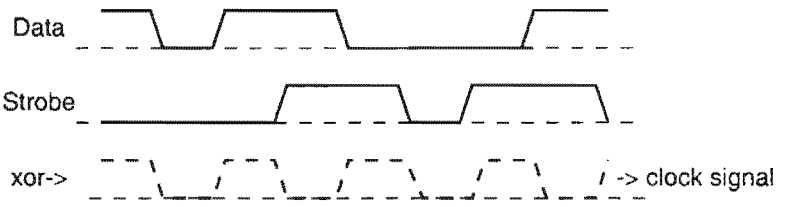


Figure 7.1: Data-Strobe Encoding

for display on the LED devices attached to the FPGA. Once this was verified, experimentation began with the FPGA's on chip Delay Locked Loops (DLL) and I/O capabilities. This involved using Xilinx vendor components in the design and thus served to test this functionality. It also tested the whether the hardware design allowed for very high-speed clock generation in terms of correct power supply decoupling and device routing. Clocks up to 266MHz were successful generated. Some implementation problems occurred in the use of the DLL components due to the use of out-of-date documentation. This was quickly resolved by using the latest documentation.

7.4.2 LVDS VHDL Test

This test design was an experiment in the use of Low Voltage Differential Signalling (LVDS). The FPGA has connected to it, 8 LVDS channels (4 in , 4 out) but contains no internal hardware serialiser/deserialiser circuitry. This design was an attempt at specifying an LVDS link in synthesisable VHDL.

Raw LVDS is simply a physical layer and provides no link capabilities. During the design period, various clock / data recovery systems were investigated for use as a link layer. Data Strobe Encoding was investigated for use as a signalling protocol because it reduces the data interference between the channels. Two LVDS channels are employed to generate Data and Strobe channels. The Strobe channel is generated from the data channel xor'ed with the transmission clock. The clock can be recovered at the receiver by simply xoring the Data channel with the Strobe channel. See figure 7.1.

Various frame detection schemes for synchronising the data were investigated. They all negatively affect the data rate of the channel by inserting framing information. Finding a reasonable implementable solution that provided a high data rate to baud rate ratio was the problem.

During the implementation of basic data transmission VHDL, serious problems were encountered. The high-speed outputs require special Double Data Rate (DDR) registers, which can be implemented using the low level components of the Virtex FPGA architecture. Unfortunately, the FPGA compiler's optimiser misinterpreted the high-level description, reducing it which caused the design to fail. At this stage, the VHDL implementation of LVDS was cancelled and a schematic entry approach was attempted.

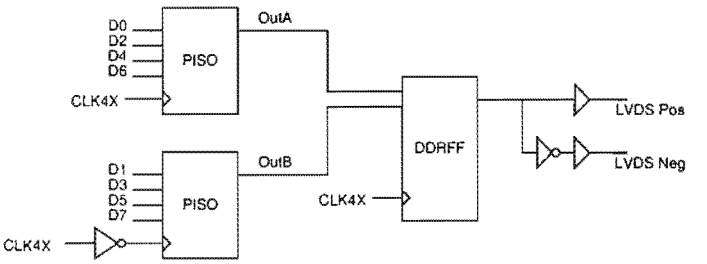


Figure 7.2: LVDS Transmitter Logic

7.4.3 LVDS Schematic Test

The implementation of an LVDS transmitter and receiver design in schematic capture is much simpler than the equivalent in VHDL. Timing constraints that are extremely tight for LVDS can be generated by simply clicking and modifying the properties of a net. There is also the added benefit that the schematic translates directly into an EDIF net-list file without the compiler trying to optimise the logic.

An example of a transmitter/receiver design in the Xilinx Application Note x233_07_062100 [x] was used as the reference for this design.

For LVDS transmission, the transmission rate far exceeds the practical internal speed limits of the device. For this reason, the high-speed sections needed special attention. The transmitter design used Parallel Input to Serial Output (PISO) components. Each PISO takes a 4-bit parallel input and a 4X-Clock from the DLL's and generates a four-stage output signal switching sequentially though each input in each stage. Two PISO devices, one positive and one negative edge triggered, have their output signal connected to two a Double Data Rate (DDR) Flip Flop which multiplexes the two signal together switching state on every edge of the 4X-Clock. See figure 7.2.

The LVDS receiver is the opposite of the transmitter, de-multiplexing the signals back to parallel output using the received clock. Note that the receiver's local clock is not synchronised to the transmitter's clock and a synchronisation barrier needed to be implemented.

The design was implemented with the FPGA system clock at 66MHz, this allowed for a theoretical baud rate of $66MHz * 4 * 2 = 528Mbits/s$. Framing information reduces this baud rate. Only basic clock and data recovery was demonstrated on the hardware to test the physical design. The FPGA connected LED devices were used to display received data in real-time.

The link was tested briefly by transmitting a pattern and displaying the recieved data on the LEDs. The results of this test showed that the synchronisation was extremely stable, the received data displayed on the LEDs remained constant even though the data rate was 528Mbits/s. The link was also observed to be very tolerant to loading and even stub connected short-circuits. A problem was observed in the test. The received pattern was not the same as the transmitted pattern. The problem was analysed with a high-speed oscilloscope and posterised to be caused by sampling the received

waveform with the wrong timing.

7.4.4 Local Bus Emulation Test

The need for the emulation of the memory controller arose from the need to test the functioning of the Bus CPLD, which implemented the Peripheral Bus Bridge and the need to access and program the FLASH memory in-system. The local bus emulation design needed to emulate exactly, the signals from the memory controller. For this reason, a good understanding of the local bus protocol was needed. The Galileo datasheet [x] and additional timing diagrams [x] provided for the device by the manufacturer proved invaluable. In addition, the timing parameters configurable on the memory controller were specified to be implemented in the design to allow various configurations to be tested.

The implementation of the design was incremental with various sections being designed and simulated to test compliance individually. The local bus emulation unit was specified to have a simple internal interface to which an intelligent module could attach to gain access to the local bus. This interface contained: a request signal and a direction signal to start a transfer, a completion strobe to indicate the completion of a bus cycle, and data input and output ports.

The design was implemented as a number of synchronous state machines that interoperated to manage the bus. A main state machine monitored the request signal and initiated a bus transfer. Firstly an address state is used to place the requested address on the bus. This is followed by either a read or write state and completed with a bus hold-off cycle. Various parallel state machines generate the cycles requires for read and write cycles as well as generating the timing and burst accesses as specified by input parameters. Various parallel processes are also implemented that monitor the states of the various state machines and generate the required control and output signals.

A simple state machine was finally written to interface the bus control logic to perform test read and write accesses to the SRAM device across the bridge. In total, ten independent parallel processes were implemented to perform the emulation of the memory controller device. The design for the basic bus emulation test was implemented in the FPGA and used the following resources: 5% of device slices, 2% of available Flip-Flops, 32% of the I/O pins and was implemented with an equivalent gate count at 9075. The place and route software optimised the design to run up to 75MHz.

7.4.5 16550 Compatible UART

The 16550 UART is the standard UART used in a PC and many embedded applications. It was thus desirable to implement a UART design that was software compatible to it. The design requirements and interface specification were taken from the National Semiconductor NS16550 UART datasheets [x]. Reference was also made to a Verilog 16550 UART design from the opencores.org Internet site for free IP cores [x]. Various design errors and flaws were identified in the Verilog design however, it

still influenced some of the implementation features. The design was specified to be implemented using the Wishbone™bus.

The design was segmented into six functional units: A transmission unit, a receiver, FIFO memories, baud rate generator, registers and control unit and a wishbone bus interface unit.

The baud rate generator is implemented as a simple down-counter that resets to a value specified at its input when it counts to zero. On reaching zero, a pulse is generated which acts as the baud rate enable. This pulse has a frequency of 16 times the wire baud rate. This is purely to increase the reliability of the receiver, which performs over-sampling and synchronises to the received start bit.

The transmitter unit was specified as a stand-alone functional unit performing parallel to serial conversion. The interface was specified to provide all the functionality required by the 16550 transmitter, including parity, bit-count and stop bit generation controls. The transmitter was implemented as a state machine with a shift-register used to serially send the data.

The receiver unit is also a functional unit that is based on a shift register to serially capture data received. The input signal is first synchronised to the system clock to prevent meta-stability in the flip-flops. A state machine monitors the state of the receive line for a start bit and then proceeds to capture the data in the format specified by its input settings. Parity, frame and break errors are detected and provided along with the received data to the interface.

The FIFO unit implements the transmit and receive FIFOs as indicated by the 16550 specification. All error information is inserted along with the received data in the receiver FIFO. Various full, empty and control signals are provided for the system control logic.

The registers unit contains the 16550 interface compatible registers and the control logic required to integrate all the components of the system. This unit has a simple bus interface to read and write the various registers. The control logic has the task of sending data to the transmitter unit from the transmit FIFO, generating interrupts and sending and receiving data from the FIFOs.

The last unit required was the WishBone bus interface, which is a simple unit that translates WishBone read/write cycles into the simple read/write interface provided by the registers unit.

7.4.6 Remote Bus Access

The remote bus access design was implemented to test various aspects of the Peripheral Bus though the FPGA across Local Bus emulation. The requirement for the design was to implement a controller on the FPGA that would allow a PC to access and modify memory devices on the Peripheral Bus through the FPGA implemented RS-232 port.

This design was implemented with three major system modules. The 16550 UART design was used to provide the serial port access and the Local Bus emulation design was used to provide access to the Peripheral Bus through the bridge in the Bus CPLD.

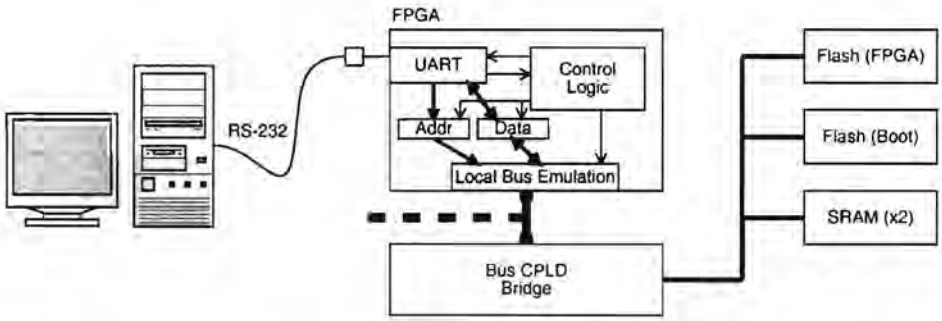


Figure 7.3: High level overview of Remote Bus Access system

The third module that was required was a control logic unit to interface the two other units and implement a protocol to communicate to the PC. A high level diagram of the system is shown in Figure 7.3.

The protocol for the RS-232 communications was specified to be a general purpose as possible, not providing the most optimal performance. The protocol specifications were:

```
Baud Rate 115200bps
Format 8 bits, Even Parity, 1 stop
PC - Communications master, FPGA - slave
Address Format - Segmented: 24 bit (16 bit segment, 8 bit address
Packet Format:
   Master - 1 Control byte followed by 1 or 2 data bytes
   Slave  - 1 Ack/Nack byte followed by 0 or 1 data bytes
Control byte (master):
   0x33 -> Read operation + 1 data byte (8-bit address)
   0xCC -> Write operation + 2 data bytes (address,data)
   0xA5 -> Set segment + 2 data bytes (16-bit segment)
Slave ACK/Nack responses:
   0xF0 -> Data acknowledge + 1 data byte
   0x0F -> Negative acknowledge + 0 data
   0x3C -> Simple Acknowledge
```

State machines diagrams were designed before the VHDL implementation in order to simplify the implementation though a better understanding of the logic sequences required by the implementation. The VHDL implementation of these state machines and the control logic driven by the states was implemented and simulated.

The final design resource utilisation in the FPGA was: 14% of Slices, 7% of Registers, 10% of LUTs and the total equivalent gate count the design was 12,887 gates. The design was optimised to run up to 72MHz.

7.4.7 Local-Bus Test

The Local-Bus test design was required to develop, verify and test the FPGA interface for implementing a slave device on the Local Bus. This is necessary in order to communicate with the FPGA from the memory controller device.

A lot of experience on the working of the Local Bus had been gained from the Bus CPLD bridge and emulation work. This design however needed to interface the asynchronous Local Bus with the synchronous internal FPGA logic.

Interfacing the asynchronous bus proved to be relatively simple in an initial working design. This basic functionality operated with the constraint that very slow timing was required from the memory controller due to the discrete sampling properties of synchronous system. What is a lot more difficult would be to re-think the design to speed up the interface. This was a low priority for this project and was left for future work.

The basic Local Bus test displayed the lower 8-bits of data written to the FPGA on the LEDs and on reading from the FPGA provided a hard-coded value on the bus.

Finally, the 16550 UART was connected to the Local Bus interface module by implementing a local-bus to WishBone bridge. This was done to test the operation of a peripheral device implemented in the FPGA from across the bus.

7.4.8 Sigma-Delta Modulator

A Sigma-Delta modulator [13] is a function that generates a digital stream of pulses from an input value. The proportion of Logic '1' outputs to Logic '0' outputs over a large sample count is equivalent to the ratio of the current input value to the maximum input value. By low pass filtering this digital bit-stream, an analogue output value can be obtained. This creates a simple Digital to Analogue Converter (DAC). The purpose of designing a Sigma-Delta modulator was to demonstrate the high-speed processing and general purpose ability of the FPGA.

The Sigma-Delta modulator was researched and a design for a simple uncompensated or signal processed converter was developed. The design consisted of two identical modulators for stereo output channels, a FIFO memory for buffering samples and a control unit that generated the timing and external interface. The sample format was specified to be 16-bits per channel in sign-magnitude format which is used for digital audio representation. The modulator was run at 66MHz with gave around 11-bits of accuracy per sample however it could be run at 133MHz to improve the signal quality. The FIFO was polled at 44100Hz so that CD quality audio samples could be demonstrated in the system.

Various revisions were required to perfect the design as problems in the initial implementation of the audio format and FIFOs existed. The design was tested initially from the host PC by a program that interfaced the Sigma-Delta converter using the Linux driver.

7.4.9 Propane UART

The 16550 UART design was unnecessarily complicated and used a lot of FPGA resources. What was required for future work where other modules were to be incorporated in the FPGA was to drop the 16550 design and specify a simpler UART design, the 'Propane UART', for eventual use in the propane system interface described in the next subsection.

The specification for the simplified Propane UART design were:

- Only 7 and 8-bit selectable communication modes
- Only 1 or 2 stop bits
- Parity modes, Even, Odd, None
- Transmit and receive FIFOs
- Interrupt on receive FIFO content size - programmable - 1,6,12,Full
- Programmable baud rate - 1100 baud to 4M baud
- Parity and Frame error detection

This design although still providing a great deal of flexibility is implementable using less logic than the 16550 UART, particularity due to the complexity of implementing infrequently used functions like break detection and 5,6 bit transmission modes.

The design implemented the WishBone interface as required by the Propane interface.

7.4.10 Propane Design

The propane system interface is a specification developed for this project to allow the processor to automatically detect and use various functional units present in the active FPGA configuration. Examples of such functional units are UARTs and the Real Time Clock interface. The propane interface aims to create a simple 'Plug and Play' environment for the FPGA designs so that operating system and boot-up software need not be modified every time a new design is loaded into the FPGA.

The Propane interface was specified and the major requirement of the system was to implement a design that met the interface specification and used the WishBone SOC bus for linking modules in the system. The Propane interface is specified to provided an interface for up to eight modules or cores to be used on the FPGA, with provision for extending this to more if required. The specification is provided in Appendix C:

The Local Bus interface from previous designs was used for the Propane design. Many problems existed with this interface and a lot of development time was required to try and obtain reliable operation. Some of the major problems experienced with

the Local Bus interface were, metastability from input signals, double read / write strobes being issued for single read / write operations and bus-hold contention. Many of these problems were only detected when implementing the Propane UART because the FIFO read and write operations are not tolerant of any extra read / write strobes and problems such as missing received characters and transmitting multiple copies of the same character were experienced. After much debugging and simulation of the interface, a stable working but slower than optimal bus interface was obtained. It was left for future work to improve the interface speed of this unit.

The various functional units were all implemented with a WishBone SOC bus interface. The Local Bus interface acted as a bridge and WishBone bus master allowing external accesses from the Local Bus to reach the addressed functional unit. No support for burst reads or writes was provided by the Local Bus interface.

The Real Time Clock (RTC) interface was specified as being part of the same functional unit as the Propane interface. The RTC device used in the project is a Xicor X1227, which provides a RTC, watchdog timer, alarms and reset and voltage monitor in a single 8-pin SOIC device. The interface to the device is the I^2C protocol [14]. This was the first logic unit required to implement the RTC functional unit. A basic I^2C master controller was designed for use by the other units. After an analysis of the X1227 I^2C interface, the master interface was specified. The interface consisted of: an address input which translated to the X1227's Clock Control Register address, 8-bit input and output data, a read/write direction control signal and Start and Done control signals.

The implementation consisted of a complicated 11-state master state machine that provided the basic major protocol state logic. Various smaller state-machines and control logic were used to generate fine grain protocol details and control the sequencing of the main state-machine. The unit interfaced the X1227 RTC at 346kHz, which is reasonably close to the rated maximum speed of 400kHz.

The I^2C protocol generated by the unit was very carefully simulated for compliance in the functional simulator. All error states and possible problems were specified to be handled in the implementation and simulated for compliance.

The RTC unit interfaced to the Propane system interface and the I^2C master controller. This allowed requests from the Local Bus, translated to the Wishbone Bus to the performed over the I^2C protocol to the X1227 RTC.

With the Propane FPGA interface design complete, adding and removing custom Propane compatible cores requires no changes to any other modules. With properly designed software, automatic detection and use of the modules provided in the FPGA is possible.

7.4.11 DSP Experiment

The Digital Signal Processing experiment was required to demonstrate the processing capabilities of the node. Two possible processing algorithms were identified for possible implementation in the FPGA: The Fast Fourier Transform (FFT) use mainly for

signal processing and Discrete Cosine Transform (DCT) used for image processing. The FFT and DCT processing cores are both supplied by Xilinx as modules for insertion into a design. On closed inspection it was found that the FFT core supplied would not be implementable in the 200,000 gate FPGA used due to specific architectural requirements of the core. It was for that reason that it was decided to use the DCT core for processing benchmarks.

The DCT DSP functional unit was implemented at a module for the Propane interface which software development to use the core simple. The DCT core was generated by the Xilinx CORE Generator with the following parameters: 16 - point DCT, 12-bit coefficient width, 9 clocks per sample latency, 8-bit data input width, 16-bit data output width, signed mathematics.

To optimise the DCT core usage, two FIFO memories were used, one for the input data storage and one for output data storage. Since the Local Bus interface is bi-directional, during write cycles to the input, no output data can be read. A control unit was designed that converted 32-bit input values from the Local Bus into four 8-bit values, serially loaded into the input FIFO. The 16-bit output values were multiplexed into 32-bit values for reading. These conversions optimised the Local Bus throughput but required extensive synchronisation logic and a lot of simulation to perfect the interface.

The full performance of the CORE was not expected to be achieved though the Local Bus interface. The DCT core used is a pipelined unit that is capable of reading an input sample generating an output on every clock edge. Thus the core at 66MHz is theoretically capable of processing $66666666.67/16 = 4.167$ million DCT transforms per second. The Local Bus interface was a great limiting factor to this performance.

7.5 Conclusions

In conclusion, various VHDL cores were specified, designed, implemented, simulated and finally implemented on the various Programmable Logic Devices (PLD) in the system. Testing configurations were used to verify the correct functioning and performance capabilities of the system. Final designs and interface specifications were developed that provide a stable and flexible base for the software environment to be developed.

Chapter 8

Software Development

Previous chapters described the Hardware and Firmware specification and design processes. The final design and development effort was related to the software for both the node and the host PC systems. The host PC was running a version of the Linux 2.4 kernel for which drivers and utility programs were developed to communicate with and test the nodes. The node software consisted of a combination of low-level assembler, embedded C and Linux application programming.

The chapter begins with a description of the software developed for the PC host system including Linux drivers, test applications and utility programs. The second part of the chapter describes the software developed for the MIPS processor on the node hardware. Various complicated design problems and their solutions are described in more detail in each of the sections.

All software developed for this project used opensource technologies including compilers, debuggers and operating systems as specified by the requirements for the project.

8.1 Linux PC Software

In order to properly use, debug and test the hardware and software on a node, PC software was required to communicate with the nodes on low-level hardware as well as high-level software interfaces. The first requirement was to interface the Galileo PCI memory controller on the nodes. Programs to communicate with the FPGA and memory over the serial port were needed to test the Bus Access firmware designs as well as the hardware. Various utility and test programs were required to provide higher level interfaces to the hardware for testing, debugging and controlling the nodes.

8.1.1 PCI Driver

The Linux kernel only allows hardware drivers access to the hardware resources in the system. Since the nodes were connected to the host PC system on the PCI bus, a driver

was required to allow user-space programs to communicate to the nodes.

A test driver was originally developed to identify the Galileo PCI device for an evaluation board that was purchased for testing the Galileo memory controller device. The main purpose of this driver was to develop an understanding of the Linux PCI driver system and provide a skeleton driver for development of the project hardware driver. This driver identified the Galileo memory controller devices in the system and registered the driver for each instance of the device found. It also performed a secondary role, which was to reset the evaluation system's processor, which was useful while experimenting on the platform.

The development of the driver for the ERPCN01 node was based on the initial driver developed for the evaluation board. The development of the driver was also progressive with features been added as required. The order in which these features are described below is a rough equivalent to the order in which they were developed. Various features that were used for testing various hardware elements were omitted in the later versions.

The first major goal of the driver was to test the Galileo memory controller. Once the hardware was in a state that the Linux host operating system identified the Galileo device, development of the driver was started.

The driver on detection of a Galileo memory controller device remaps the memory windows provided by the device into the host system's memory space. This effectively allows the physical memory on the node to be accessible directly by the driver. Firstly, the driver was modified to perform write operations to the SRAM device on the node. A correct read from that device would verify the operation of the Galileo device, Local Bus - Peripheral Bus Bridge and SRAM device. During this stage many problems were encountered which were gradually solved as a greater understanding of the Galileo device and the issues associated with a PC environment. One particular problem that took a long time to rectify was caused by the PC's PCI BIOS reprogramming the Galileo's PCI registers, adjusting the PCI memory map without reprogramming the internal maps. This caused memory access errors and PCI retry operations, which effectively locked up the host PC in certain situations.

Following the testing of devices on the Peripheral Bus, the SDRAM memory needed to be initialised and tested. Since the Galileo's internal registers are memory mapped to the host, full control of the device is possible without the on board processor's intervention. The SDRAM memory controller needed to be programmed to enable the SDRAM and to setup the device timings. The initialisation sequence and timing settings from the SDRAM's datasheets were used as a reference for this. A test sequence then proceeded to write and verify the SDRAM to test its functionality.

Once access to physical memory devices on the project hardware from the driver was functional, a device interface for user programs was implemented. A Linux character device interface was chosen for access to the physical memory on the board. This would allow user programs to read and write various memory locations with a file like interface. In addition, the Linux kernel provides a memory map file access extension which allows a user space program to access the memory identically to the way it

accesses its local memory.

A driver interface and the four memory windows on the Galileo device were allocated device names:

```
/dev/parnodectlX     - 63:(X)       - driver control
/dev/parnoderegX     - 63:(X+8)     - Galileo registers
/dev/parnodesdramX   - 63:(X+16)    - SDRAM memory
/dev/parnodedevsX    - 63:(X+24)    - Chip-selects 0..2
/dev/parnodebootX    - 63:(X+32)    - Boot device and CS 3
```

These devices are UNIX devices with a major number of 63, indicating the driver ID and minor numbers identifying the sub-device referenced. The 'X' is replaced by a number from '0' to '7' and indicates the node number in the system. The 'parnode' reference stands for 'Parallel Node', which is how the system identifies the ERPCN01 nodes.

The size of each of the Galileo windows is different, however in some cases, the window is not large enough to address the entire memory space of the addressed device. For this reason, PCI windows can be offset into a device address space to access these areas. One of the primary reasons for the driver control interface was to provide an interface to modify these offsets.

The final addition to the Linux 'parnode' driver was a network emulation layer to provide a virtual network between the local PC operating system and the Linux operating system that was implemented on the MIPS processor.

Two approaches to this networking emulation were attempted. The first approach was to emulate a full Ethernet network including Ethernet MAC emulation and Address Resolution Protocol support. This work was based on the Linux PCI-skeleton networking code for NE2000 compatible PCI cards. The transmit and receive buffers that are provided by a physical network card are emulated by using shared memory between the host and target board. The basic principle of operation was to place a packet ready for transmission into the shared memory buffer and signal the remote side that a packet has arrived. The remote driver will then read this buffer and insert the packet into the networking layer as if the network card had just received it. This approach worked well initially for the low level Ethernet emulation and ARP / ICMP protocols however, TCP connections were being ignored. After an extensive debugging effort, a solution was not found and it was decided to try the alternative approach that was to emulate a Point-To-Point link.

For the Point-To-Point link, the basic concepts were the same as for the Ethernet link except that no Ethernet framing information, MAC addresses or ARP messages are sent across the link. The kernel knows the IP address of the target machine and less processing is performed as no Ethernet Frame decoding has to be performed. This driver used the Linux Virtual Point-to-Point(TUN) and Ethernet(TAP) device drivers as a reference base. This driver was implemented and after debugging and fixing various problems, a working driver was obtained.

Various work was performed to try and improve the performance of the virtual networking link. The initial driver used the unused upper bank of the SRAM on the node as a shared memory buffer. The host operating system did not use this device in its system memory map. For this reason, no other programs or data would be affected by using this memory. The problem however was that it was an 8-bit device with slow timing. To try and increase access speed, a 32kb block of SDRAM main memory was reserved early during the Linux MIPS kernel memory management initialisation. This was reserved for the networking driver. Both the host PC and board processor had much greater bandwidth to the 32-bit SDRAM memory and the performance increase justified the effort.

8.1.2 Bus Access and FLASH programmer

The Bus Access design in the FPGA implemented a serial port and protocol to which a PC could interface in order to read and write devices on the Peripheral Bus though the Local Bus. The first application to be developed was a speed-test program to test the system by reading and writing the SRAM device as fast as the serial port allowed. During the development of this program, the routines for handling the communications protocol were developed.

One of the primary reasons for developing the Bus Access design was to allow the FLASH memory on the board to be programmed without the need for a working processor or memory controller. A suite of programs was written to perform actions using the Bus Access design.

The next program to be developed was a FLASH identification program. It was required to read the FLASH identification code from the FLASH devices on the board using the common FLASH interface, which the Flash devices supported. The Flash interface is memory mapped in the FLASH devices and allows the programming, erasing and controlling of the device. The protocol was implemented using the specifications given in the FLASH datasheets. This involved sending a sequence of commands to the device and reading back the result. The successful reading of the FLASH ID would also verify the operation of the device as it was the first test to interface the devices.

A similar program to erase the FLASH memory was written based on the Get_ID program. Finally, two programs, 'program_flash' and 'fast_program' were developed to program the FLASH memory. They both implemented different programming algorithms as specified in the datasheets [x]. The 'fast_program' program was finally used to program the FPGA Configuration FLASH memory device with an FPGA configuration. The standard FLASH programming routines in the 'program_flash' program involved many more many read/write operations, which over the serial link it was calculated to take more than 3hours to program the device. Finally, routines to read the FPGA configuration files in Intel MSC-86 format were written in order to program the FPGA FLASH device with configuration data. The FLASH programming operations were proven successful as the read-back from the device was identical to the file programmed into the device.

It is possible to implement the FLASH programming routines in the FPGA but the time taken to implement the design was not worth the effort. The main method for programming the FLASH in the final system would be from the host PC system across the PCI bus. This method would be orders of magnitude faster. The Bus Access design merely proved a useful development and testing tool.

8.1.3 Debugging and Utilities

A full suite of debugging, testing and utility programs were written for the host PC Linux system for interfacing the ERPCN01 nodes. The major programs are described briefly below.

Flash-PCI

The Flash-PCI utilities are a set of programs to test and program the FLASH memory devices on the boards. They talk to the board using the 'parnode' Linux driver and use the same programming and identification routines as the Bus Access software.

UART 16550 Test

In order to test the 16550 VHDL design in the FPGA from across the Local Bus, a simple program was written to memory map the FPGA through the Linux 'parnode' driver. This allowed the registers of the UART in the FPGA to be modified by simply reading and writing to a pointer. This created a very simple and fast utility to directly control the UART in the FPGA.

FPGA Configuration

One of the major aims of the project was to enable FPGA configuration at run-time in the system. Run-time basically means that the process occurs while the rest of the system is running. The software to program the FPGA is fairly simple. The FPGA configuration file is decoded and programmed into the SRAM configuration buffer. The Bus CPLD device is then triggered to program the FPGA by accessing the system interface it provides.

Audio Test

The Sigma-Delta firmware design in the FPGA was tested by sending raw audio samples to the FPGA into the FIFO buffer. A simple loop was written that polled the FIFO state in the FPGA and inserted data as required. The input data was taken from the system 'stdin' (standard input) which allowed audio data to be 'piped' into the program. Most of the debugging and fixing was involved with overcoming the read latencies and buffering the input to prevent the FIFO from emptying.

Utility Programs

A number of utility programs were developed to aid various system operations and functional requirements. Some of these programs were:

cpu_reboot Accesses the Bus CPLD system interface to reboot the processor

cpu_reset Accesses the Bus CPLD system interface to reset the processor

cpu_run Accesses the Bus CPLD system interface to release the processor from reset

dev_dump Programs a binary file into a memory location in the CS 0..2 PCI window

dev_read Reads from a memory location in the CS 0..2 PCI window

map_cs1_devs Instructs the 'parnode' driver to map the PCI window CS0..2 to CS1

reset_maps Instructs the 'parnode' driver to set the PCI windows back to their original state

sdram_read Reads from the SDRAM memory

setup_sdram_boot Programs the memory controller to boot the processor from SDRAM

setup_sram_boot Programs the memory controller to boot the processor from SRAM

One last Unix program, 'dd' was used extensively to dump binary files into the board memory though the device driver.

8.2 MIPS Software

The major software development effort focused on the software written for the MIPS processor in the project hardware. Four distinct sets of software were developed during the course of this project, test programs to verify the operation of the processor and system, the diesel boot-loader software to initialise and interact with the processor, the Linux operating kernel porting effort and various Linux applications.

Before any programs for the MIPS processor could be developed, a development environment needed to be setup on a development PC. The development tools that were used are all GNU opensource software and consisted of: *binutils* - Binary utilities for creating, modifying, extracting and viewing compile object files and archives. *GCC* - The GNU C Compiler. *GDB* - The GNU Debugger including DDD and Insight. *glibc* - The GNU standard C Library. *newlib* - An embedded glibc alternative with no operating system calls. Three sets of tools were configured for different processor configurations and feature usage. These were: MIPSel - little endian 32-bit MIPS compatible, MIPS64el - little endian 64-bit MIPS compatible and MIPSel-Linux - little 32-bit MIPS compatible for Linux applications. The MIPS Linux tool-chain had many problems and issues at the time of use and pre-compiled, tested versions of the compilers were used to ensure correct operation.

8.2.1 Verification Programs

The first programs to be developed were used to test the compiler tool chains and gain a greater understanding of the tools and the MIPS instruction set. The code written for these tests was in MIPS assembler language and was used to boot the processor and initialise the Galileo memory controller. This code was simulated but never run on the processor.

The first program to be executed on the processor was designed to be run from SRAM. The program configured processor for basic operation and a main loop incremented a 64-bit counter at a specific memory location starting from value '0'. During this test, the processor was running un-cached which meant that the physical memory location was constantly updated and could be read from the host PC across the PCI bus. This was used to verifiy the operation of the processor.

The first programs written in 'C' were then developed to test the compiler tool chains. An initialisation function in assembler language was used to initialise the processor, setup the stack and clear the program heap. This is required to start execution of a 'C' program. The GNU term for this file is the 'crt0' file, which is used to initialise all programs and is required to be custom written for embedded systems. The first 'C' program tested implemented the same counter as the previous counter in assembler. This was followed by a program that interfaced the 16550 UART in the FPGA to echo received characters.

One last assembler program was written to test the SDRAM memory. A value was written to the SDRAM while the program was running from SRAM to verify that the processor to SDRAM interface of the GT64115 was working.

8.2.2 Diesel Boot-Loader

Before work on the Linux kernel could begin, a boot loader and debugging platform was required which would also have the responsibility of setting up the hardware system for Linux and executing it.

The processor initialisation code from the previous programs was used as a base for the Diesel initialisation. Various additional checks such as endianness and Galileo configuration were added to prevent the program from executing under an incorrect hardware environment. Interrupt handling routines were written to provide error messages when the program execution caused an exception. Tests to determine from what location the program was running were performed to allow various features to be enabled or disabled. One such feature was that processor caches should only be used when running from SDRAM memory. The processor cache controller was reset and the various routines to enable a 'C' environment were performed before executing the main program.

The Diesel Boot-Loader was designed to be simple and made easy to add new features. A serial port communications interface was designed so that changing the serial port functionality (in the FPGA) would not affect the reset of the system. A shell inter-

face was written to provide a user interface to the system. This interface implemented a 'plug-in' style architecture to allow various shell commands to be added into the system to provide functionality. The basic shell functions provided were:

boot Execute a program from a user provided address, typically used to start Linux

cache Enable/disable processor caches as well as changing the cache modes

cp0 Allows access to the processors Co-Processor 0 (cp0) for system control

help Provide information on the use of the shell and various shell commands

memory Display the memory controller's configuration in an easy to understand way

port Perform read or write operations to a specified address (processor address)

propane Display the core modules present in the Propane system interface

reset Perform a processor reset

rtc Display and modify the date and time in the on board Real Time Clock

The diesel boot loader was later modified to provide a test platform for running various benchmarks in a 64-bit environment without an operating system.

8.2.3 Linux Kernel

The porting of 64-bit little endian Linux to the ERPCN01 hardware platform took the majority of the software development time and effort. During the development period, various versions of the kernel were experimented with and much of the kernel memory management, filesystem and architecture specific code was under speculation.

The porting effort from the start was a very big task, which needed to be broken down into smaller units in order to manage the work load and understand what was required. The first task was to take the kernel sources and add in support for the ERPCN01 platform into the MIPS64 architecture port. The next task was to fix the kernel source tree, adding configuration settings and fixing problems to allow the kernel to compile. Then the task of debugging this code on the actual hardware was started. Drivers for the serial port, virtual networking and sigma-delta audio processor were then written. One last task was to implement an initialisation RAM disk from which to boot the Linux system.

The first task was to implement architecture specific code involved developing interrupt, timer, system and error handling interfaces for the kernel. The bus error handling code was taken from the Silicon Graphics 'Indy' port as the processor internal operation was identical. A low-level interrupt handler based on the 'Indy' port was written in assembler that differentiated between timer and other system interrupts. The reason for handling the timer differently was in order to reduce the latency of this high

frequency interrupt. The main high-level interrupt handler was written to handle interrupts from the various devices in the system. The MIPS processor only provides five hardware interrupt lines, however each interrupt can be cascaded i.e. the Galileo memory controller has a single interrupt line to the processor but has 21 possible interrupt sources. The interrupt handler will respond the the Galileo interrupt by requesting the cause of the interrupt from the device. A virtual interrupt table was specified for the direct and cascaded interrupts as shown in Table 8.1.

Routines were written to allow the enabling and disabling of any interrupt by number. This would automatically mask or unmask cascaded interrupts were necessary. Handlers for the Galileo memory controller interrupts and the Propane interface interrupts were written to handle specific details concerning the two interfaces. Setup routines to initialise the memory system prior to the starting of the generic MIPS memory management system were needed. These functions cleared the Linux stack, detected how the system was configured and setup memory regions as per the requirements of the MIPS memory interface. This is where the shared memory buffer was reserved for the virtual networking device. A driver was implemented to setup the MIPS internal timer to act as the Linux kernel timer. Much of the timer code was devised from the 'Indy' port. A driver to interface the Real Time Clock interface provided in the Propane system interface was developed. This was used to set the Linux clock and keep it synchronised. Finally, routines to reset and reboot the processor were provided as required by the Linux kernel.

The task of getting the kernel sources setup for the new configuration and compiling was the next major task. It was discovered that the 64-bit little endian support in the Linux kernel was not complete. During the process of setting up the kernel configuration options in order to compile the required source code and setup the correct memory maps, the various missing sections of code were written as 'stub' code to allow the kernel to compile. These 'stubs' would need to be written at a later stage during the debugging of the kernel when required. The initial kernel version that was worked on was the Linux 2.4.9 kernel that was the latest version when development started. It was found that various changes had been made to the kernel, which had not been corrected in the MIPS port. A lot of development time was spent on fixing the MIPS port to allow the kernel to compile.

After a fixing the problems preventing the kernel from compiling, the task of debugging, fixing and implementing 'stub' routines as started. The Diesel boot loader program was used to execute the kernel, which was placed into SDRAM memory by the host PC over the PCI bus. Starting from the kernel entry point, an iterative process of inserting debugging checkpoints and break-points was started. Each initialisation function on the Linux kernel was verified as debugging progressed. As 'stub' functions were encountered, they were replaced with the required code to implement the function. One of the first tasks was to get the kernel console drivers operational which required a working serial port driver. When the serial port driver was working and the console linked to it, it allowed kernel-debugging messages to be echoed back to the PC system from the serial port. As functions were verified, the debugging check-points

Table 8.1: Virtual Interrupt Allocation

Interrupt	Cascaded IRQ	Description
0	no	Software Interrupt 0
1	no	Software Interrupt 1
2	no	Galileo GT64115
3	no	FPGA
4	no	PCI / Bus CPLD
5	no	Bus CPLD
6	no	Config CPLD
7	no	MIPS Timer Counter
16	2	CPU Memory Address Bounds Error
17	2	DMA Memory Address Bounds Error
18	2	PCI Host Access Error (Memory Error / Parity etc)
19	2	DMA-0 Complete
20	2	DMA-1 Complete
21	2	DMA-2 Complete
22	2	DMA-3 Complete
23	2	Timer 0 Interrupt
24	2	Timer 1 Interrupt
25	2	Timer 2 Interrupt
26	2	Timer 3 Interrupt
27	2	PCI Read Parity Error
28	2	PCI Write Parity Error
30	2	PCI Abort Termination
31	2	PCI Retry Expire
32	2	Power Management Request
33	2	PCI Interrupt 0
34	2	PCI Interrupt 1
35	2	PCI Interrupt 2
36	2	PCI Interrupt 3
37	2	PCI Interrupt 4 (remote debug request)
48-55	3	Propane Interface Interrupt 0-7
48	3	Real Time Clock
49	3	Typically Propane UART (should auto-detect)
63	-	Last interrupt

and break-points were removed to allow the kernel to boot further.

Some major problems were encountered during this process as soon as the cache and memory management sections of the processor were enabled. Problems in the cache caused strange and unpredictable operation and a lot of time was spent reading back the SDRAM and comparing it to the original binary and Linux System Map. A lot of time was spent on the Linux MIPS IRC channel talking to the developers from SGI and other companies in order to solve these problems.

Various small goals were assigned during this development period such as getting the MIPS timer operational, enabling system interrupt handling and fixing memory management problems. The task of identifying the cause of the problems that resulted from the memory management issues prompted a trace of program execution though the kernel filesystem, drivers and process handling code. The problem was finally tracked down to a problem with the Translation Look-aside Buffers (TLB) in the MIPS memory management low-level routines. A quick fix was implemented to fix the problem but this had some performance problems associated with it. The proper fix was left up to the Linux MIPS developers who identified that a problem existed but could not state when it would be fixed. Development on the getting the rest of the system operational was a priority and the memory management was left in this state.

Once the kernel was operational to such a state that it required the 'init' program to be executed to boot the system, work started on implementing an initialisation RAM disk. An initialisation RAM disk is a filesystem based in memory that the kernel uses for its root filesystem in order to run just enough programs to enable access to a larger, possibly remote file system. For the embedded system, this filesystem needed to be linked into the kernel object code by the linker, which would provide pointers to data location for the kernel. Code to implement this needed to be written and utility programs to package the RAM disk as a linkable object file were written. The Linux MIPS applications in the RAM disk are fortunately totally unaware of the underlying hardware as the kernel provides all the interfaces. For this reason, a standard Linux MIPS RAM disk file can be obtained or generated that will work in any Linux MIPS kernel as long as the endianness of the files is correct.

Debugging the kernel although complicated by the caches was reasonably possible with the use of debugging messages and monitoring memory from the host. When interrupts were enabled however and processes started to be spawned, any piece of code in the interrupt handler and system call interface could be called at anytime from any process or interrupt routine. Debugging in an environment where you are not sure what is running when or whether the currently executed function will complete before an interrupt occurs and the system starts executing other code before returning and completing the function became very difficult.

Debugging user-land processes became even more complicated because they execute in virtual memory which may or may not have existed in physical memory and probably not in contiguous blocks. User space programs were tracked by inserting debugging code in the kernel system call entry and exit points to try and deduce what operating system calls were being executed.

A major problem in the entire system was that the kernel was running in 64-bit mode and user space programs in 32-bit mode. This is primarily due to the fact that the Linux compilers were unable to produce 64-bit Linux user space programs at the time of development. Every system call had to be evaluated to make sure that it was 32-bit safe. This was implemented as a 32-bit compatibility layer that was mostly provided by the standard MIPS kernel but a few extra handlers had to be implemented. The most common problem of running a 32-bit program on a 64-bit kernel is that the 'C' size of pointers and 'long' variables are different and thus data-structure sizes may differ. This can cause memory overruns to occur resulting in unpredictable errors.

After a very extensive debugging period, the Linux kernel was successfully able to execute a shell program, giving console access to the system. This allowed the development of further Linux application software for the system to be started.

Following the successful implementation of Linux on the ERPCN01 system, drivers were written to implement the sigma-delta audio interface and virtual networking interfaces. The sound card driver that was needed to provide an interface to the sigma-delta DSP was developed from existing Linux drivers but did not implement most of the 'ioctls' (I/O controls) provided by the interface. The driver was written to provide just enough functionality to 'pipe' audio data from the application, through the driver to the FPGA. A ring buffer was implemented in the driver to prevent buffer under-run problems and the FPGA FIFO state was polled at a constant rate and new data provided when necessary. Interrupt operation was tried but was unsuccessful and the time taken to fix this could not be justified.

The virtual networking driver was implemented to provide a TCP/IP communications link between the host PC and the node. Two versions of the driver were implemented as described in the section on the Linux host PC driver. The specifications for the virtual Point-to-Point networking link, which was eventually sucessfully implemented were:

- Two 'mail box' shared memory windows would be used for transmit and receive buffers
- Each 'mailbox' will have a field indicating the size of the message packet and an area of memory for the packet.
- The signalling between the two processors will be done using interrupts. All required information will be present in the 'mailbox' for the receiving side to use.
- Presently, no acknowledge of receipt is implemented.
- The interface will emulate a TCP/IP Point-to-Point link, no other low-level protocols will be usable.
- The virtual interrupt number 33 in the Linux MIPS kernel was allocated for the transmit interrupt and 34 used for the receive interrupt. See Table 8.1.

- The SDRAM memory range reserved for shared memory communications was 32kb at address 0x00C00000.

The network link enabled the use of Network File System (NFS) between the host and the node. The standard Linux NFS drivers that come as part of the source code were used unmodified and the Linux 'mount' program was used to mount the host's exported filesystems to the node.

8.2.4 Linux Programs

Developing user-space programs for Linux MIPS is as simple and changing the compiler used to compile programs for Linux on a standard PC. Once the Linux kernel was operational, software development could focus on benchmarking and system testing. What follows is a short description of some of the various programs used for testing and benchmarking the Linux MIPS platform.

mpg123

One of the first processing intensive programs to be tested on the processor was an MPEG audio decoding program called mpg123. The Linux audio driver written to interface the sigma-delta converter was used as the output device. MPEG 1 Layer 3 audio files were successfully demonstrated on the system.

FFTW

FFTW is a collection of fast C routines to compute the Discreet Fourier Transform in one or more dimensions. Provided with the library is a test programs that performs one or multi-dimensional transforms given a set of command line options.

whetstone

A 'C' converted version of the Whetstone Double Precision Benchmark. This is a simple benchmark to provide a performance measure of both floating point and integer performance of a processor.

fastdct

Fastdct is a program to compute the Discrete Cosine Transform using the algorithm from IEEE signal proc, 1992 #8, Yugoslavian authors. This program was used to benchmark the processors DCT performance to compare with the FPGA performance.

tcpserver test

A simple TCP server running on port 37 (time) was written to test the networking code in the kernel. This program used simple Unix sockets to listen on the port. The program was first tested on a standard Intel PC running Linux before being compiled for the MIPS processor.

PVM

From the PVM manual: PVM is a software system that enabled a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource, or a "parallel virtual machine".

PVM is a program that runs on various computer systems and provides transparent communications between computing machines. A master program is responsible for managing the application and requests processing clients to be initiated and provided them with data. The PVM software has the responsiblility of executing the programs on the remote machines as well as performing communications with data conversion where machines have different data representations. Because each machine can be different, versions of the client programs must be compiled for each of the target architectures.

The first problem experienced with PVM was trying to make the application to compile and execute. After sorting out a lot of problems with the PVM source code, a compiled version of PVM was obtained. After a lot of work, PVM executed on the system and opend network connections but failed to communicate with the host PC. This problem could not be resolved in a reasonable time frame.

miscellaneous

A simple http server program was compiled and tested on the platform.

Various network client programs such as 'ping' and 'ftp' were used to verify the operation of the networking interface. Network sniffing programs such as 'tcpdump' were used on the host system to monitor network communications.

Unix and Linux programs such as 'mount', 'cat', 'ls' and 'ifconfig' were used extensively on the system.

Chapter 9

Testing and Verification

Testing and verification was important in order to demonstrate that the hardware and software performed according to the requirements. This chapter describes the testing and verification procedures performed to test the project firmware and software. The hardware verification process was described in chapter 6.

9.1 Firmware Verification

9.1.1 Config CPLD

The Config CPLD firmware design in VHDL was tested incrementally during the design phase though the use of simulations. Each sub-section of the design was tested after implementation to verify correct logic behaviour. The design consisted of elements that produced real-time delays in excess of 100ms from a 66MHz clock. These were extremely large simulations and thus the delays were reduced considerably for the simulated tests. A full simulation at 100ps resolution would require 10,000,000,000/66,666,666.667*8,500,000 = 1.275 billion steps.

Once the design had been fully simulated, corrected and verified, the task of testing the design in physical devices was initiated. Initially, the design was modified to keep the processor-reset signals asserted while enabling the functionality of the rest of the system. This allowed the testing of the reset delay functionality without affecting the processor.

Once it was verified that the reset delay counters operated correctly, the processor initialisation section of the design was enabled. An oscilloscope was used to verify the bit-streams generated from the CPLD as well as checking the timing conformance of the various CPU interface signals. Various unforeseen problems such as incorrect bit-ordering and inverted signals from the dip-switches were identified and corrected.

Finally, the processor initialisation sequences were verified and the final processor reset logic was enabled. An oscilloscope was used to monitor the processor's SysAD bus for activity which would indicate that the device had been initialised. This con-

firmed the correct functioning of the Config CPLD design.

9.1.2 Bus + Control CPLD

The Bus CPLD design was extensively simulated to verify the correct operation of the local-bus interface before the design was implemented in the CPLD. Any problems in the design could cause bus contention, which could interfere with the operation of other devices on the bus.

A 500MHz oscilloscope with a PC interface was used to assemble a timing diagram of the Bus CPLD's interaction with an emulated memory controller in the FPGA. See 7.4.4. The FPGA simulated four byte burst transfers across the Bus CPLD bridge to the SRAM and Flash Devices.

The only major problem encountered during the testing of the Bus CPLD with the memory controller. The device did not respond to any requests on the Local Bus. During an investigation of the problem, every control signal on the Local Bus was captured to make sure the memory controller was producing the correct signals. Two problems were discovered, the memory controller was not generating a chip-select signal and the Bus CPLD design specified an incorrect pin location for a Local Bus signal. The memory controller problem was tracked down to a problem in the memory controllers register configuration, which was modified by the PC hosts PCI BIOS.

Once memory behind the Bus CPLD on the Peripheral Bus Bridge was accessible and working correctly, the Bus CPLD design was accepted as meeting the requirements.

9.1.3 FPGA Testing

Many designs were implemented in the FPGA during this project. Many of them reused components from previous implementations such as the UART module, Local Bus interface and Propane system interface. Each module was tested individually and proven before use in other designs.

The initial tests of the FPGA were required to verify that the hardware design was correct and that the FPGA was functional. These tests output signals to the LEDs to provide a visual verification of the design.

The Local Bus emulation testing and verification was performed mostly in the simulator, with functional and timing simulations being performed. The Local Bus interface was required to be correct in order to verify the operation of the Bus CPLD. The design was worked on until it satisfied all the specifications of the Local Bus interface in the simulations. A captured trace of the physical bus signals was also acquired with the digital oscilloscope.

The next FPGA design to be tested was the 16550 UART core. The initial tests were performed first on the transmitter unit. A continuous stream of characters was output from the FPGA. Once the baud rate generations and transmitter unit were functioning as specified, the characters were received and verified using a serial port on the

host PC. The receiver was the next unit to be tested. The transmitter was connected to the receiver unit so that received characters were retransmitted or 'echoed' back to the PC. The correct reception of transmitted characters verified the UART transmitter and receiver units. The final test of the UART was the generation of a state machine that interfaced the 16550 UART bus interface, setting up the registers initially, then polling the receiver. On reception of a character, it was read from the UART, incremented by one and sent to the transmitter back to the PC. This verified the operation of the 16550 UART design.

The Local Bus and 16550 UART designs were linked with control logic that implemented a protocol for communicating over the serial port. This design was verified in stages. The first goal was to enable reading and writing to the SRAM in order to verify the designs operation. A digital oscilloscope was used to verify the assertion of the correct signals on the Local and Peripheral busses. Once the read and write operations were working, the design was shown to be working correctly, the last task was to verify that the FLASH devices in the system could be identified and programmed. Successful reading and writing the FLASH device proved that the design was working correctly. During the FLASH programming, over 700,000 characters were correctly transfered over the UART link without a single error occurrence.

The Local Bus device interface was tested with the Galileo memory controller the master of the Local Bus. Transactions on the Local Bus were requested from the host PC across the PCI Bus. During testing, the interface was refined to resolve problems such as write cycle synchronisation. The final design was verified when the FPGA interface was functioning as specified and the physical signals on the Bus were verified with the oscilloscope.

The sigma-delta modulator design was only verified by the quality of the output signal. The first tests produced noise from the output, which vaguely contained hints of the presence of a signal. This was found to be a problem with the modulator design and corrected. The next problem was that the digital audio format was being interpreted incorrectly and the signal was being 'clipped' and inverted which produced very poor quality sound masked with a heavy noise component. Once this was identified and fixed, the audio quality output equated to AM radio. Various problems causing the FIFO buffer to under-run were fixed and eventually, high quality audio output was achieved with only very slight modulation noise audible at very low output volume levels.

The final testing on the system was concerned with the verification of the Propane system interface design and its associated modules. The first task was to verify the read/write access to the interface and then the operation of each device. For these tasks, a program on the host PC was written to test the Propane interface across the PCI Bus. The verifying of the Propane interface found a problem with the Local Bus interface that had been adapted for this design. A lot of work was required to find and solve this problem. This difficulty was mainly due to the fact that certain problems only occurred with some of the Propane modules. The interface was finally corrected and verified.

Each of the modules implemented for the Propane interface including the Propane UART, Real Time Clock interface, Sigma-delta processor and Discrete Cosine Transform processor were all individually tested. The test program on the PC host identified each of the modules present in the Propane interface and called functions to test them. Each module was tested and fixed to meet the specifications before the system was declared to be working correctly.

9.2 Software Testing

Through most of this project, software was used to test hardware and firmware designs. The only software component that was tested with other software was the Linux operating system.

Testing Linux was performed primarily to identify problems with the port in order to fix them. The first program tested on the system was BusyBox, a program containing various common Unix utility functions built into a single file for embedded systems. The 'init' part of the program emulates the Linux Init program which is used for system start-up. During the initial stages of getting Linux to execute user programs, the 'init' program was modified to do various system calls to try and find the causes of problems in the port. One major problem that was identified and solved with this program was that the 'sys_info' system call was not 32-bit safe and a 32-bit version sys32_info was created to address the problem.

Various networking and audio programs were used to verify the virtual networking link and sigma-delta sound card interface. The networking testing was a considerable effort caused mainly by the problems in implementing the Ethernet network emulation. The TCP/IP Point-to-Point link caused a lot less problems. The proving test for the networking link was to mount a filesystem from the host PC on the Linux MIPS kernel via the NFS protocol. NFS operation was stable and reliable. A small TCP server program was written and run on the Linux MIPS kernel to test connections initiated by the host to a server on the Linux MIPS system. This proved successful and meant that it would be possible to implement programs to communicate and exchange information over TCP/IP for parallel processing.

9.3 Benchmarks

Various benchmarks were performed on the processor and FPGA to analyse the performance of the prototype system and compare it to other reference systems. Most of the benchmarks on the processor were run under the Linux operating system. Due to limitations in the tools available, only 32-bit programs could be created. This limited the instruction set available to the compiler for optimisation. The pre-compiled libraries used were created for stability and only used the MIPS-I instruction set which reduced the performance of the system.

Table 9.1: Benchmark Results

Program	MIPS-32 200MHz	x86 Athlon 1GHz	Ratio
Whetstone	93.5MIPS	588.2MIPS	79.5%
FFTW n=64	113.87 mflops	590.94 mflops	96.3%
FFTW n=256	93.99 mflops	622.077 mflops	75.5%
FFTW n=1024	103.3 mflops	604.22 mflops	85.5%
FFTW n=65536	24.39 mflops	127.7 mflops	95.5%
FFTW n=131072	23.33 mflops	98.23 mflops	118.8%
FFTW 1024x512	29.334 mflops	265.2 mflops	55.3%
DCT - 2 point	621.12k/sec	5.556M/sec	55.9%
DCT - 4 point	318.47k/sec	3.125Mwhich/sec	50.9%
DCT - 8 point	151.975k/sec	1.47M/sec	51.7%
DCT - 16 point	74.184k/sec	719.424k/sec	51.6%
DCT - 32 point	34.842k/sec	303.951k/sec	57.3%
DCT - 64 point	16.102k/sec	149.701k/sec	53.8%
DCT - 128 point	7.404k/sec	73.099k/sec	50.6%

The results of various benchmarks are shown in Table 9.1.

Figure 9.1 shows the normalised performance of the MIPS processor at 200MHz compared to the 1GHz AMD Athlon (normalised to 200MHz). Past 1024 points, the smaller cache on the MIPS processor makes the performance drop off.

The final benchmarks performed on the platform were to test the FPGAs processing capabilities compared to the MIPS processor and a standard PC. Two possible algorithms were identified for testing in the FPGA. The FFT function was the first common function identified and is used in a wide variety of signal and image processing applications. The Discrete Cosine Transform (DCT) is a common function using in image and audio processing, most notably in JPEG and MPEG processing. The FFT logic core provided by the manufacture was not compatible with the particular FPGA that was chosen. This left the DCT function for testing. A DCT module was developed using the manufacture provided core. This was integrated into the Propane interface and a driver to access it was written for Linux.

The results of the 16-point DCT transform benchmark for the MIPS processor, FPGA and 1GHz Athlon processor were: MIPS CPU - X DCTs per second, FPGA - 450,000 DCTs per second, Athlon - 700,000 DCTs per second.

Note, the speed of the FPGA was greatly limited by the FPGA bus interface, which requires a re-design.

The results for the network speeds were: 1.2MB per second transfer speed over the NFS link. This was mainly limited by the CPU overhead involved. The ICMP speed reached 2.374MB/s with the 'ping' program. Raw coping of data across the PCI bus reached 32.768MB/s which is the maximum speed of the PCI Bus.

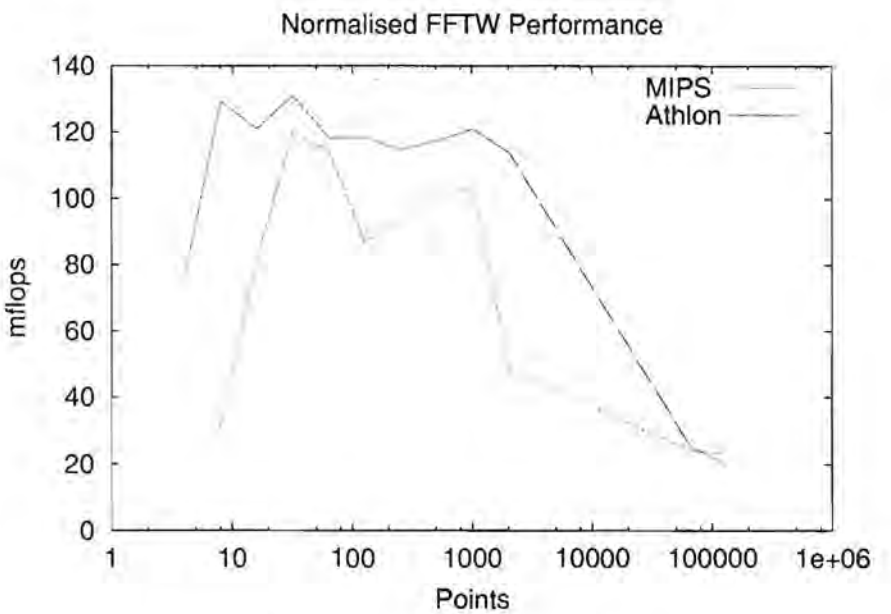


Figure 9.1: Normalised FFTW performance

Chapter 10

Conclusions and Future Work

This chapter presents the conclusions of the project as taken from the results of the hardware, firmware and softwares design, development and implementation. The various requirements that were specified and the manner in which these were achieved is described. After the conclusions, is a section describing possible future work that can be performed on the hardware system.

10.1 Conclusions

A hardware prototype platform with accompanying system software was developed for the application of re-configurable hardware parallel processing research.

The hardware platform provides all that is needed for a stand-alone microprocessor system with configurable logic. A 64-bit MIPS processor is provided for general purpose processing and control, a memory controller and SDRAM memory provides high-speed high capacity memory required for high speed computing with large data sets. Sufficient FLASH memory is provided for processor booting and FPGA configuration and an SRAM device is provided as an FPGA configuration cache. The FPGA provides system functions and the ability to implement application specific hardware processing routines and a simple interface for the processor to access it is provided. The system provides Real Time Clock, processor configuration and Reset management devices that provide support to the hardware and software environment. LVDS channels are provided by the FPGA for high-speed inter-node communications via a dedicated interface.

The software developed for the system provides a base for future research in parallel processing with the system. A boot loader and system test application was developed to execute the Linux operating system as well as providing a simple base for real-time application development. The Linux operating system was ported the to MIPS processor and provides a software environment that is familiar to people with Unix and Linux experience. The benefits of the Linux operating system were that standard Linux programs simply require a recompile and binary files from other Linux

MIPS systems are directly supported.

The software for the hardware platform also required the support of programs and drivers on the host PC running the Linux operating system. These programs provide debugging, testing and essential functions for loading and executing code on the processor.

The PCI Bus connection to a host PC provides a very fast and simple interface to the project hardware and allows firmware designs to be tested directly from the host without having to interface them via the MIPS processor. The PCI Bus interface also provides for real-time access to the platform memory while the processor is in operation, which simplifies debugging and allows for shared memory communications.

The MIPS processor was demonstrated to perform well despite the restrictions under which the benchmarks were performed, namely running in 32-bit mode under Linux without using only the MIPS-I and MIPS-II instructions. The processor performed 75-100% of the normalised speed of an AMD Athlon 1GHz processor with a 266MHz Front Side Bus. Given that the AMD processor has 256kb cache and advanced features such as branch prediction, the MIPS processor performed well and could possibly outperform the AMD processor if its full set of features is deployed.

The FPGAs DCT algorithm is pipelined and can theoretically perform over 4 million transforms per second at 66MHz. The external Local Bus interface however limited the performance as data could not be provided and read back fast enough. The FPGA DCT benchmark still produced reasonable results at around 450,000 transforms per second. This was still in the order of 5.8 times faster than the most optimised algorithm running on the MIPS processor. The FPGA benchmarks have shown that even a simple un-optimised algorithm can have huge performance advantages when performed by the FPGA.

10.2 Future Work

The focus of the project was the development of a hardware platform with supporting software for re-configurable parallel processing research. In order for the platform to be of use for parallel processing, applications need to be developed to make use the hardware. A developer skilled in Linux and FPGA programming could develop an application to use the platform as a base for experimenting with various algorithms. The PCI bus can be used for point-to-point communications and with more work, a shared infrastructure could be developed. A communications link can also be implemented over the LVDS channels to provided dedicated high-speed communications between nodes. Experiments with the PVM software package can be performed and work can be focused porting it to use the high-speed custom communications channels.

Further work can also take the form of optimising the hardware platform. On the provided hardware, better bus interface logic can be experimented with including DMA and VUMA support. Future work can also analyse the strong and weak points of the current design and improve upon it. Given that the design developed during

this project was a prototype aimed mainly for implementing a demonstration platform, future work can be focused on optimising the performance. Faster processors, faster and wider buses and deeper memory are features that should possibly be implemented.

Future work can also look at partial reconfiguration of the FPGA. Although the current platform was design to permit this, none of the software tools used supported this feature of the FPGA.

Bibliography

[1] E. Minty, R. Davey, A. Simpson, D. Henty. Decomposing the Potentially Parallel. *Edinburgh Parallel Computing Centre, The University of Edinburgh* http://www.epcc.ed.ac.uk/epcc-tec/documents/decomp-course/

[2] D. Sweetman. See MIPS Run. *Morgan Kaufmann Publishers*, San Francisco, California, USA, 1999

[3] IDT. 79RC64574 79RC64575 User Reference Manual. *Integrated Devices Technology.* 2975 Stender Way, Santa Clara, CA 95054. March, 2000

[4] Xilinx. Virtex-E 1.8V Field Programmable Gate Arrays. *DS022-2 (v2.1).* April, 2001

[5] NetBSD: http://www.netbsd.org/

[6] Linux: http://www.linux.org

[7] RedHat Software: eCos operating system. http://www.redhat.com/products/ecos

[8] L4: L4 on MIPS R4x00. http://www.cse.unsw.edu.au/~disy/L4/MIPS/index.html

[9] Galileo Technology: GT-64115 System Controller for RC4640, RM523X, and VR4300 CPUs. *Revision 1.11.* April, 2000

[10] Xilinx: XC95144XL High Performance CPLD. Version 1.2. November, 1998

[11] H. Johnson, M Graham. High Speed Digital Design: A Handbook of Black Magic. *Olympic Technology Group*, Redmond, WA

[12] T. Wallis. EMC For Product Designers. *Newnes*, Linacre House, Jordan Hill, Oxford OX2 8DP

[13] G. Noriega. Sigma-Delta A/D Converters - Audio and Medium Bandwidths. *RMS Instruments,* 6877 Goreway Drive Mississauga, Ontario, L4V-1L9, February 1996

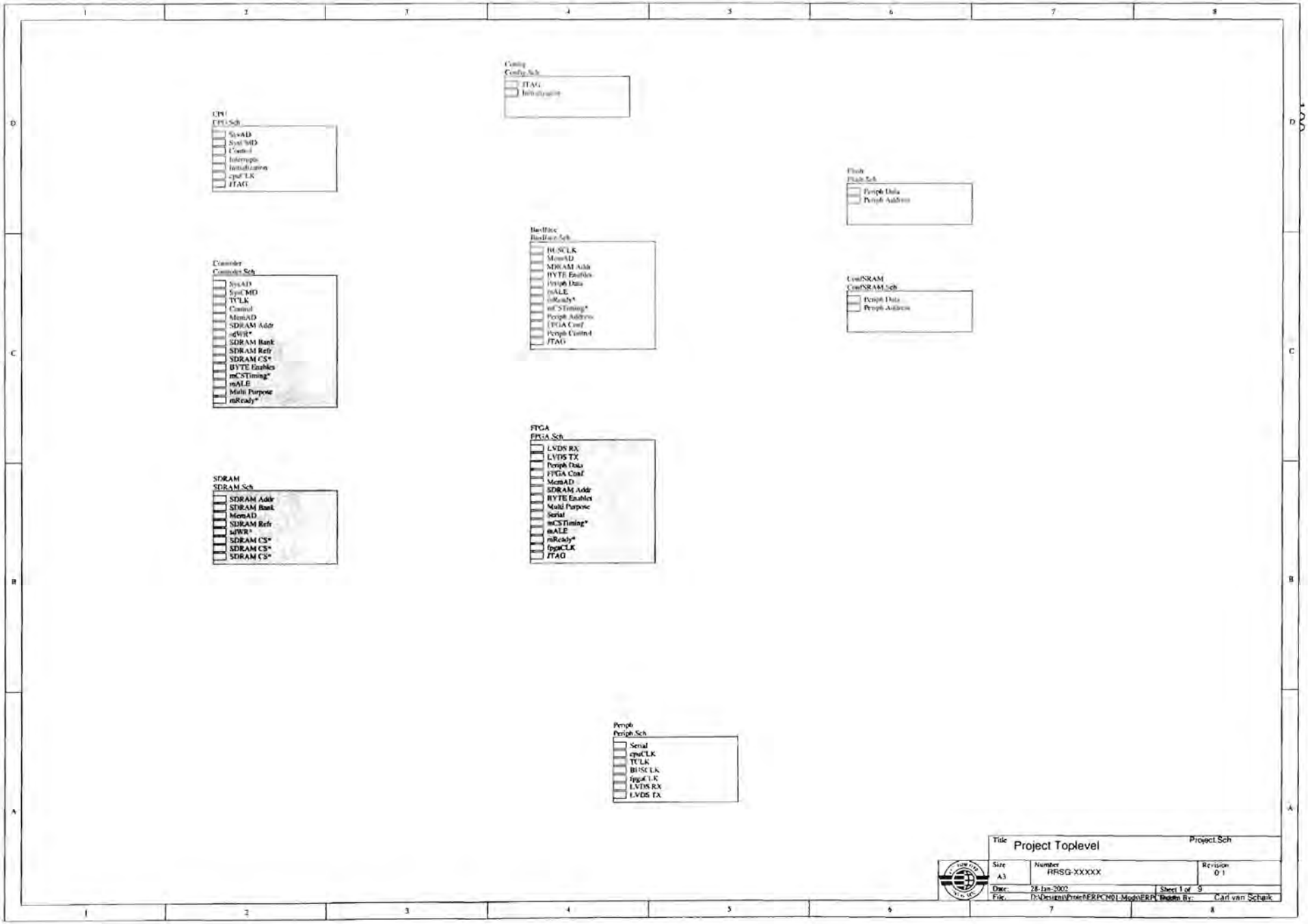
[14] Philips Semiconductors. The I^2C Bus Specification.

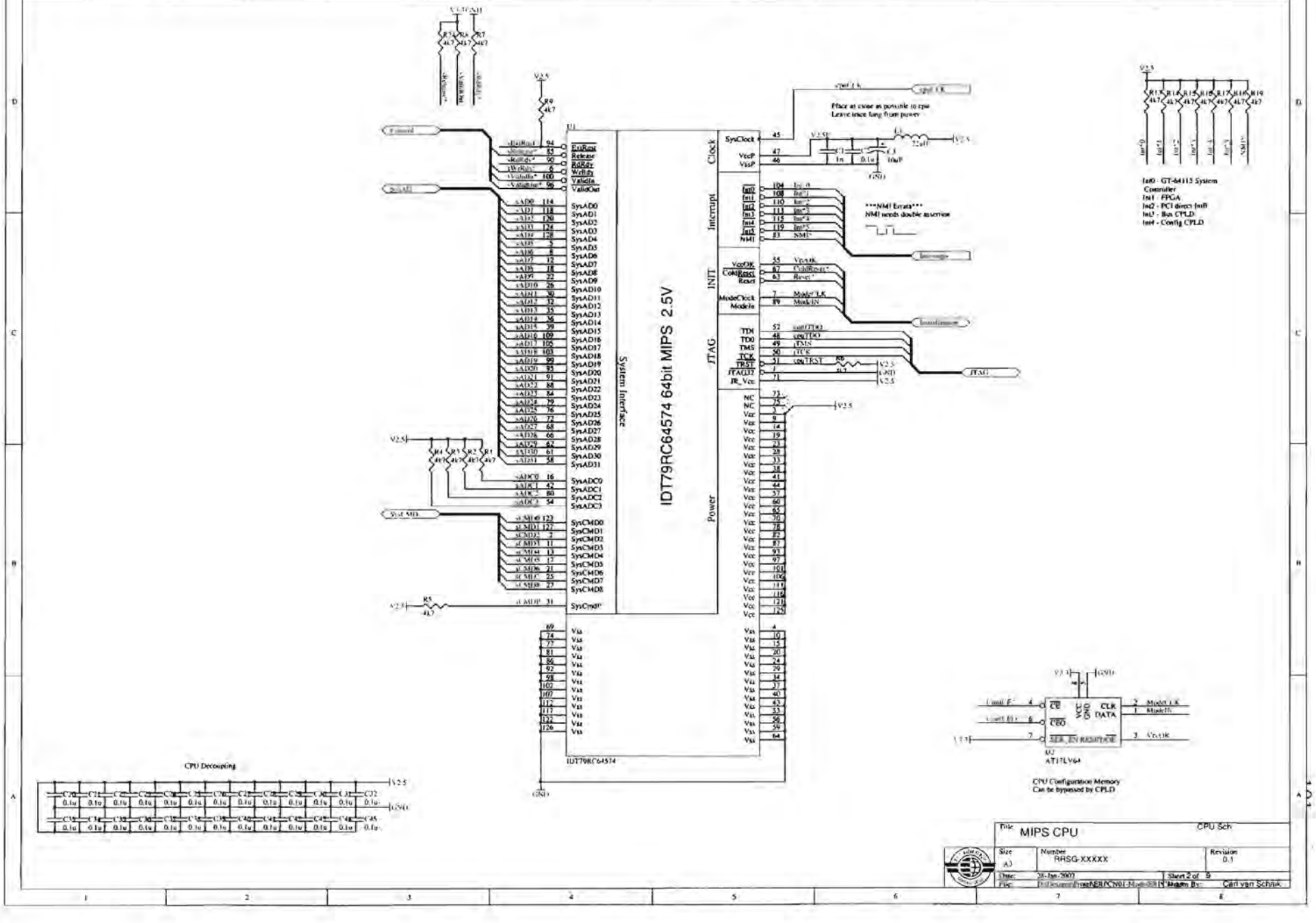
[15] Xilinx: Configuring Virtex FPGAs from Parallel EPROMs with a CPLD. *XAPP 137* March 1, 1999 (Version 1.0)

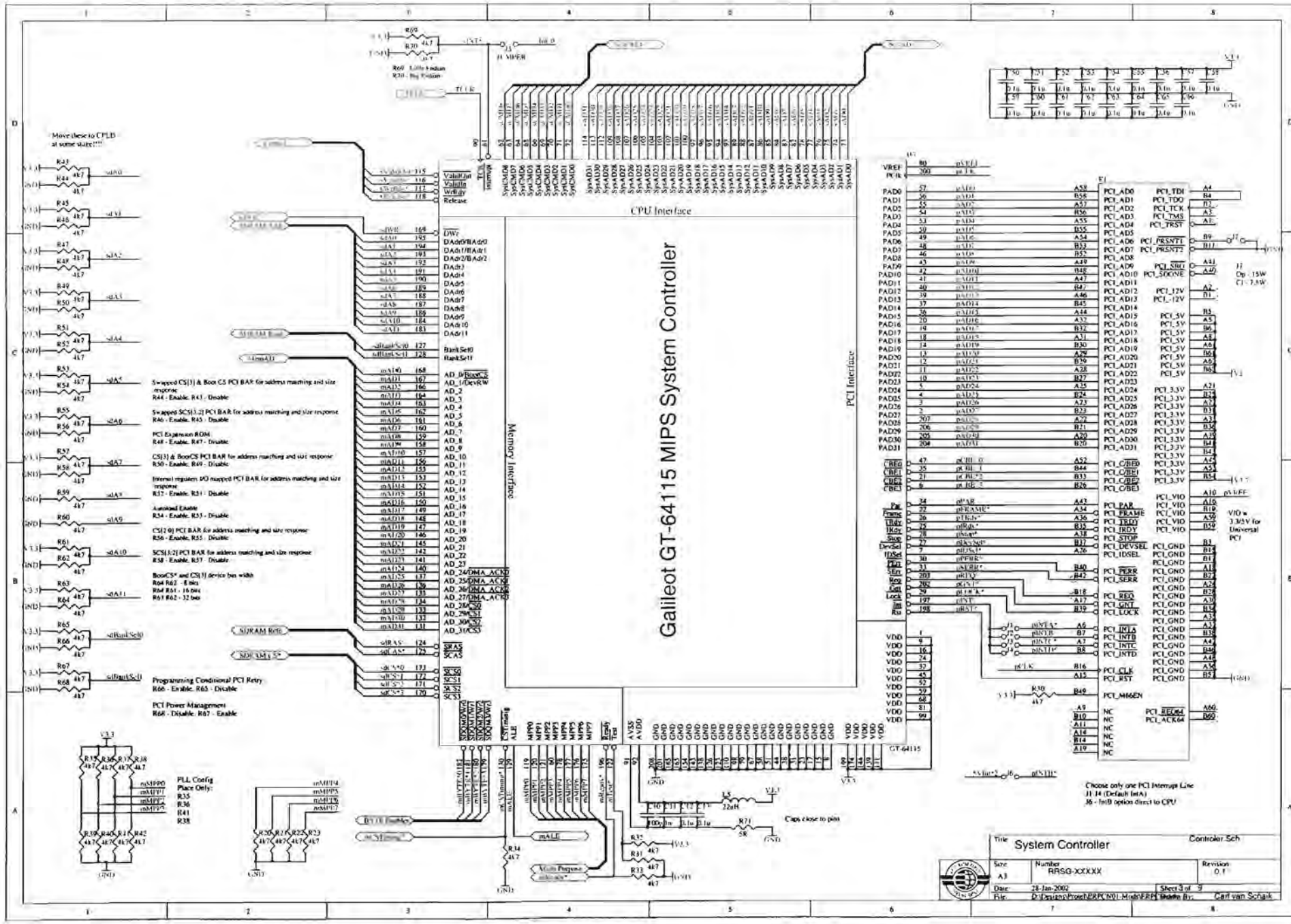
[16] Xilinx: Virtex FPGA Series Configuration and Readback. *XAPP138* (v2.3) October 4, 2000

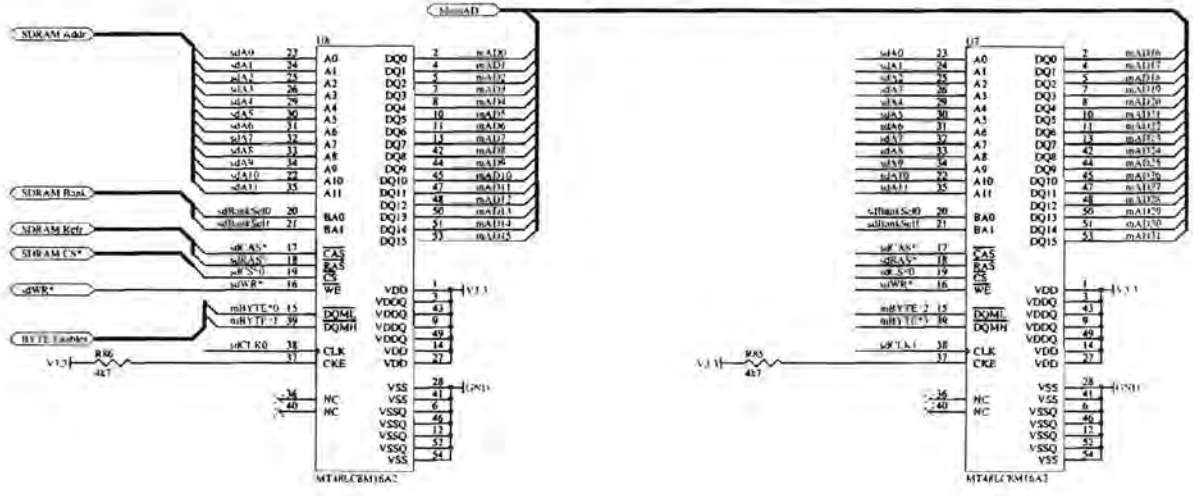
Appendix A

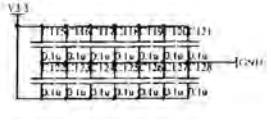
Schematics and PCB


Config
Config.Sch
JTAG
CPU
CPU.Sch
SysAD
SysCMD
Control
Interrupts
Initialization
cpuCLK
JTAG
Flash
Flash.Sch
Periph Data
Periph Address
BusIface
BusIface.Sch
BUSCLK
MemAD
MBRAM Addr
BYTE Enables
Periph Data
mALE
mReady*
mCSTiming*
Periph Address
FPGA Conf
Periph Control
JTAG
Conn/SRAM
Conn/SRAM.Sch
Periph Data
Periph Address
Controller
Controller.Sch
SysAD
SysCMD
TCLK
Control
MemAD
SDRAM Addr
sdWR*
SDRAM Bank
SDRAM Refr
SDRAM CS*
BYTE Enables
mCSTiming*
mALE
Multi Purpose
mReady*
FPGA
FPGA.Sch
LVDS RX
LVDS TX
Periph Data
FPGA Conf
MemAD
SDRAM Addr
BYTE Enables
Multi Purpose
Serial
mCSTiming*
mALE
mReady*
fpgaCLK
JTAG
SDRAM
SDRAM.Sch
SDRAM Addr
SDRAM Bank
MemAD
SDRAM Refr
sdWR*
SDRAM CS*
SDRAM CS*
SDRAM CS*
Periph
Periph.Sch
Serial
cpuCLK
TCLK
BUSCLK
fpgaCLK
LVDS RX
LVDS TX
Title
Project Toplevel
Project.Sch
Size
A3
Number
RRSG-XXXXX
Revision
01
Date:
28-Jan-2002
Sheet 1 of 9
File:
D:\Designs\Proiect\RRPCN01-Mod\RRPC
Drawn By:
Carl van Schaik

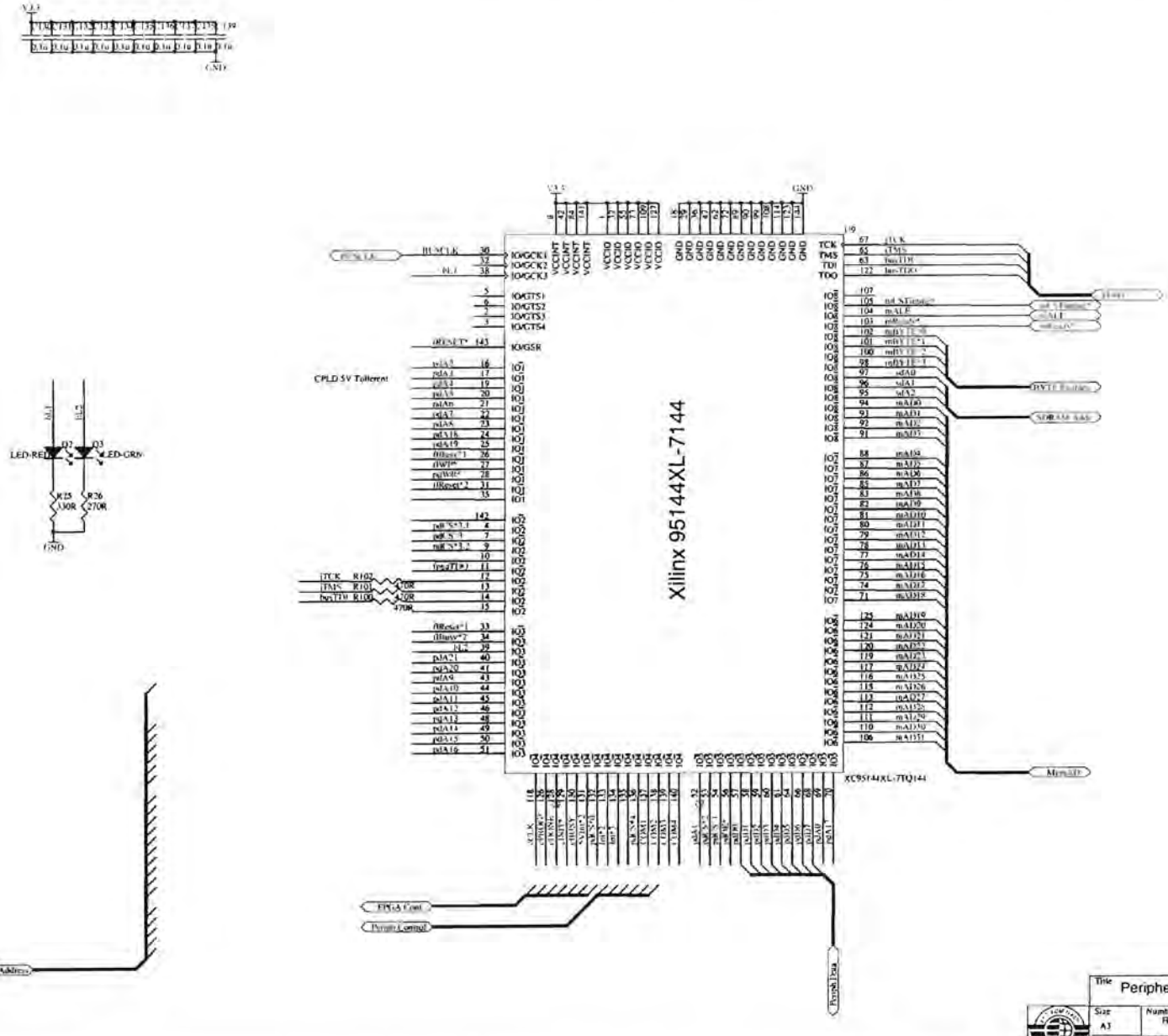

IDT79RC64574 64bit MIPS 2.5V
System Interface
Clock
Interrupt
INTT
JTAG
Power
IDT79RC64574
Place as close as possible to cpu
Leave space free from power
NMI Errata
NMI needs double assertion
Int0 - GT-64115 System Controller
Int1 - FPGA
Int2 - PCI direct Int0
Int3 - Bus CPLD
Int4 - Config CPLD
CPU Decoupling
AT17LV64
CPU Configuration Memory
Can be bypassed by CPLD
MIPS CPU
CPU Sch
RHSG-XXXXX
Carl van Schaik

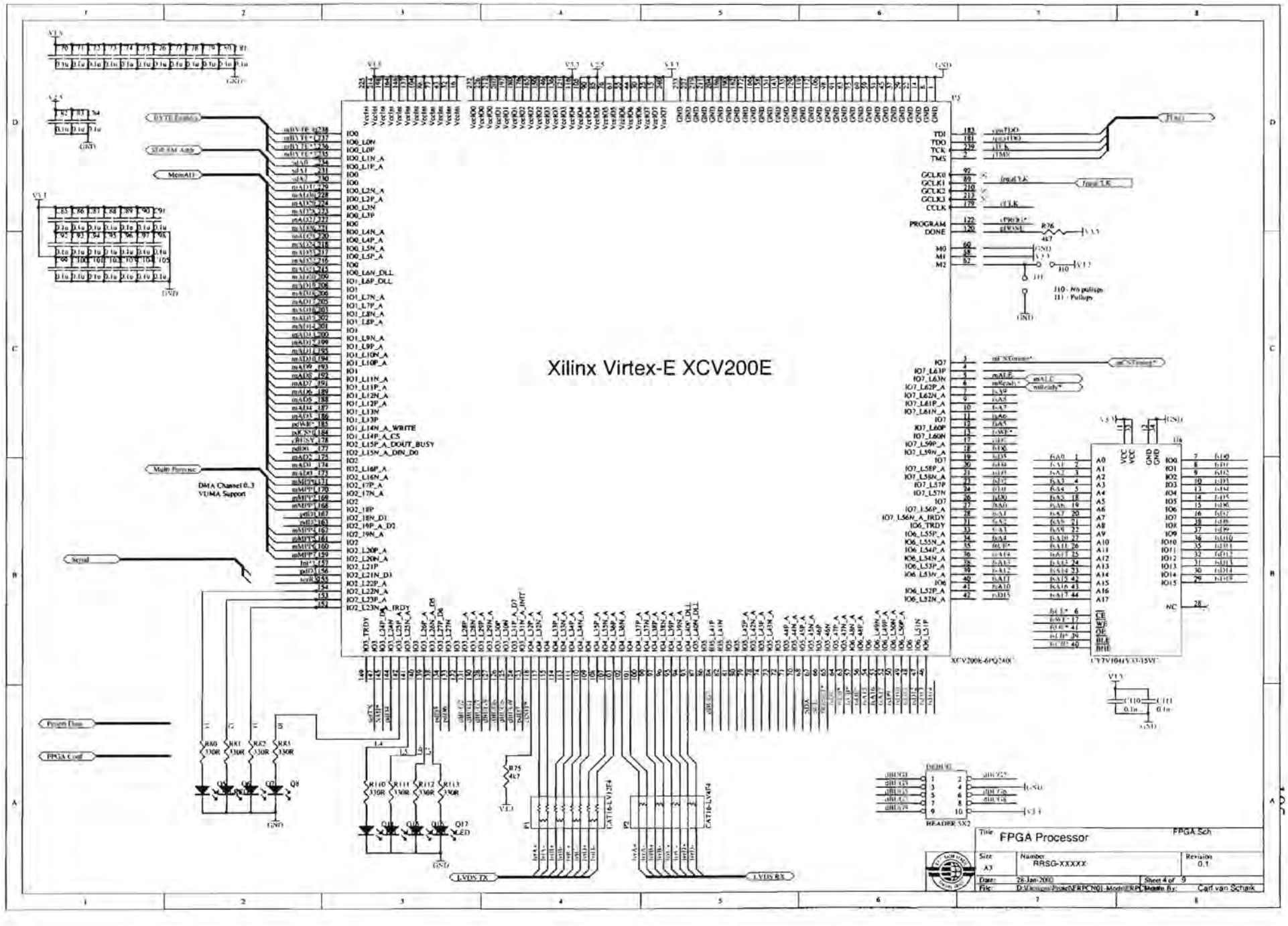

Galileot GT-64115 MIPS System Controller
CPU Interface
Memory Interface
PCI Interface
Move these to CPLD at some stage!!!!
Swapped CS[3] & BootCS PCI BAR for address matching and size response
R44 - Enable, R43 - Disable
Swapped SCS[3:2] PCI BAR for address matching and size response
R46 - Enable, R45 - Disable
PCI Expansion ROM
R48 - Enable, R47 - Disable
CS[3] & BootCS PCI BAR for address matching and size response
R50 - Enable, R49 - Disable
Internal register I/O mapped PCI BAR for address matching and size response
R51 - Enable, R51 - Disable
Autoload Enable
R34 - Enable, R33 - Disable
CS[2:0] PCI BAR for address matching and size response
R56 - Enable, R55 - Disable
SCS[3:2] PCI BAR for address matching and size response
R58 - Enable, R57 - Disable
BootCS* and CS[3] device bus width
R64 R62 - 8 bits
R64 R61 - 16 bits
R61 R62 - 32 bits
Programming Conditional PCI Retry
R66 - Enable, R65 - Disable
PCI Power Management
R68 - Disable, R67 - Enable
PLL Config
Place Only:
R35
R36
R41
R38
Caps close to pins
Choose only one PCI Interrupt Line
J1 J4 (Default Int A)
J6 - Int9 option direct to CPU
Title: System Controller
Controller.Sch
Size: A3
Number: RRSG-XXXXX
Revision: 0.1
Date: 28-Jan-2002
Sheet 3 of 9
File: D:\Projects\Prosel\RRPCN01-Mod\RRPCN01-Mod.SchDoc
Made By: Carl van Schaik

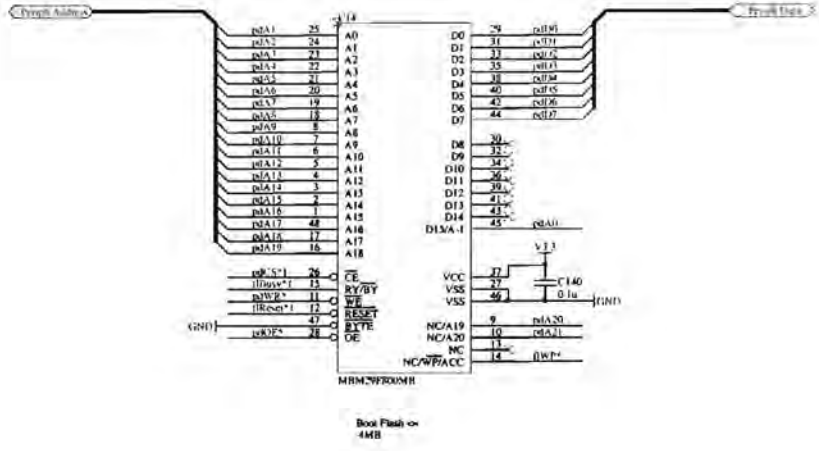


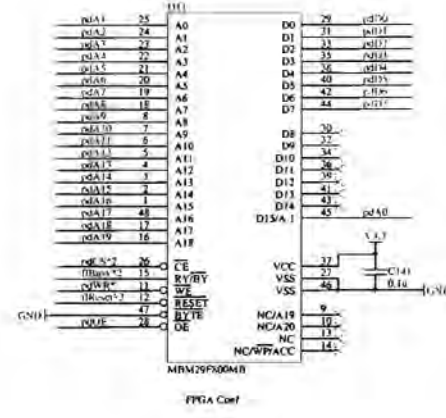


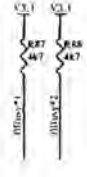
Title	SDRAM.Sch		
Size: A3	Number:	Revision:	
Date: 20-Jan-2002	Sheet 6 of 9		
File: D:\Designs\Phase4\ERPCN01-Main\SDRAM.Sch	Drawn By: Carl van Schaik		


Xilinx 95144XL-7144
XC95144XL-7TQ144
LED-RED
LED-GRN
R25 130R
R26 270R
GND
CPLD 5V Tollerant
Periph Address
Periph Data
FPGA Cont
Periph Contol
Peripheral Bus Interface
Busface.Sch
Size A3
Number RRSG-XXXXX
Revision 0.1
Date: 28-Jun-2002
Sheet 5 of 9
Carl van Schaik

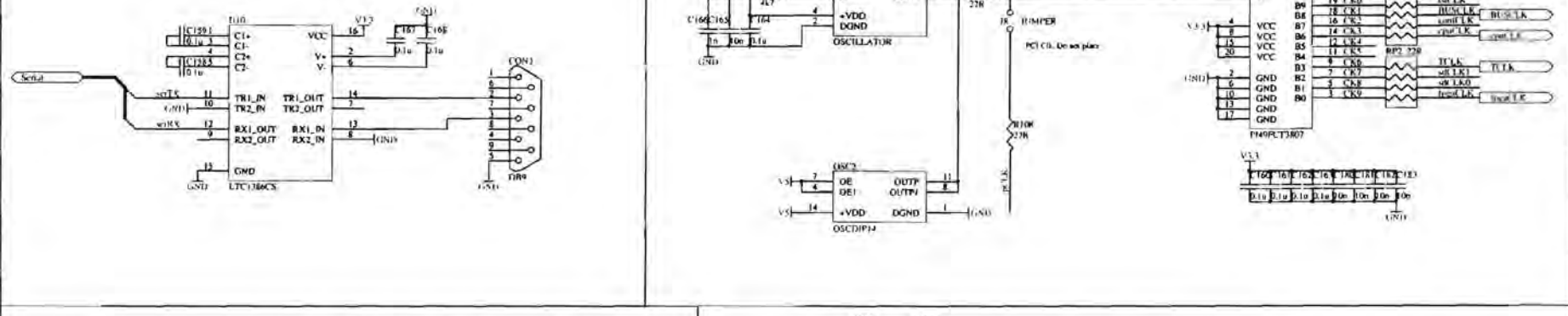

Xilinx Virtex-E XCV200E
FPGA Processor
FPGA Sch
Size
A3
Number
RRSG-XXXXX
Revision
0.1
Date:
28-Jan-2003
Sheet 4 of 9
File:
Drawn By:
Carl van Schaik
DMA Channel 0..3
VUMA Support
LVDS TX
LVDS RX
LED
XCV200E-6FG240
CY7V1041V33-15VC
HEADER 5X2

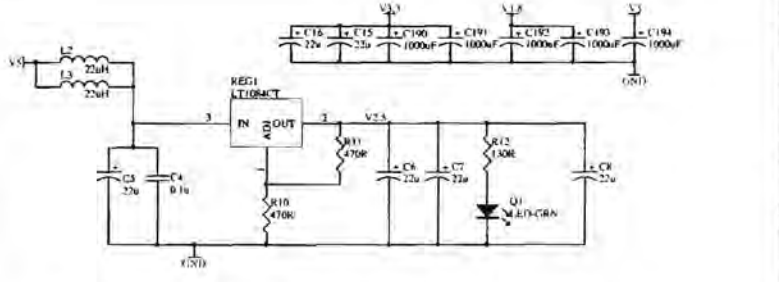


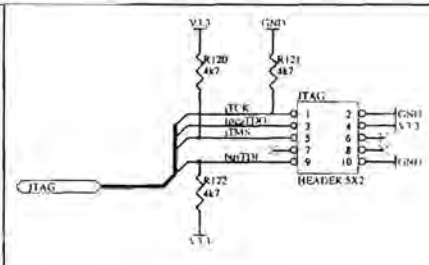


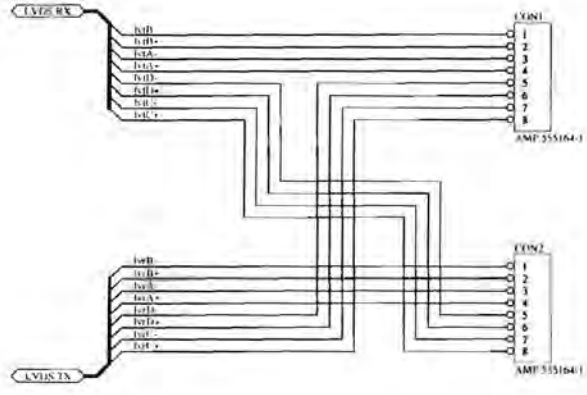


Title: FPGA and Boot Flash			Flash Sch
Size: A3	Number: RRSG-XXXXX		Revision: 0.1
Date: 28-Jan-2002	Sheet 7 of 9		
File: D:\Designs\Prive\ERP\CN01-Mods\ERP[illegible].dsn		Drawn By:	Carl van Schaik

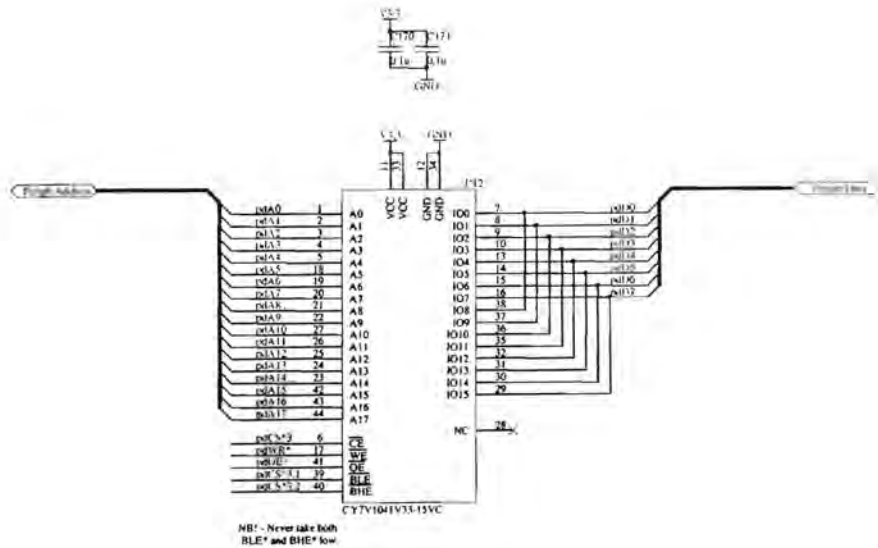




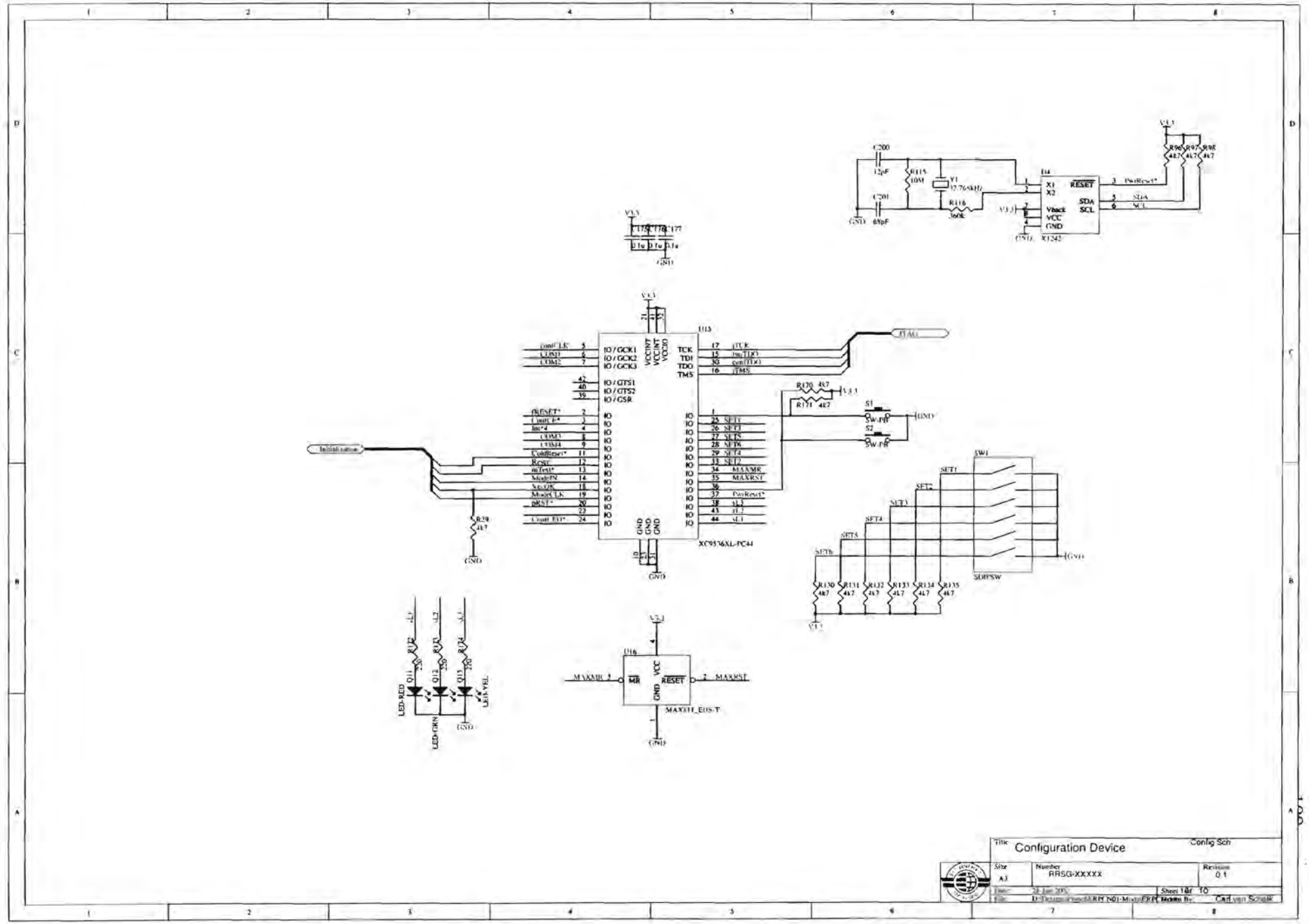


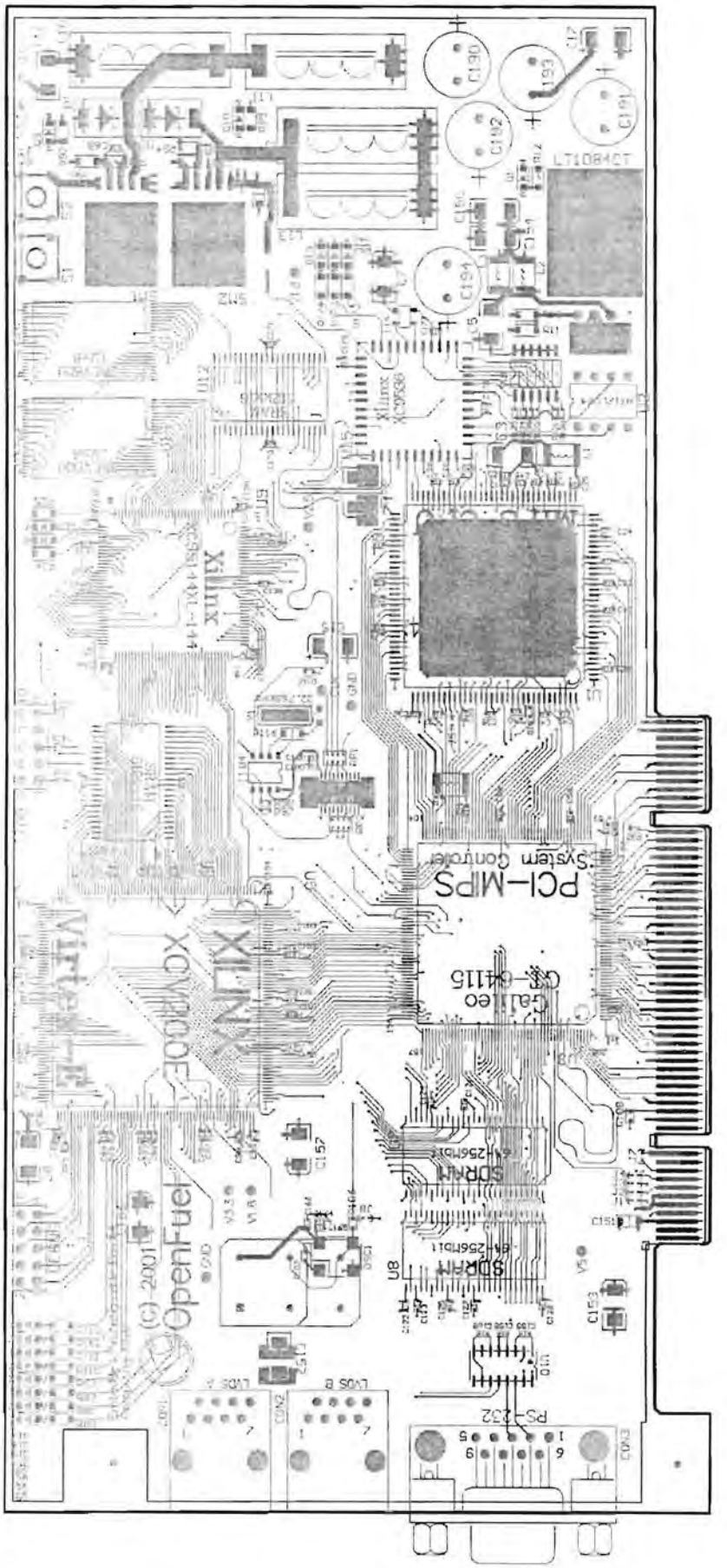


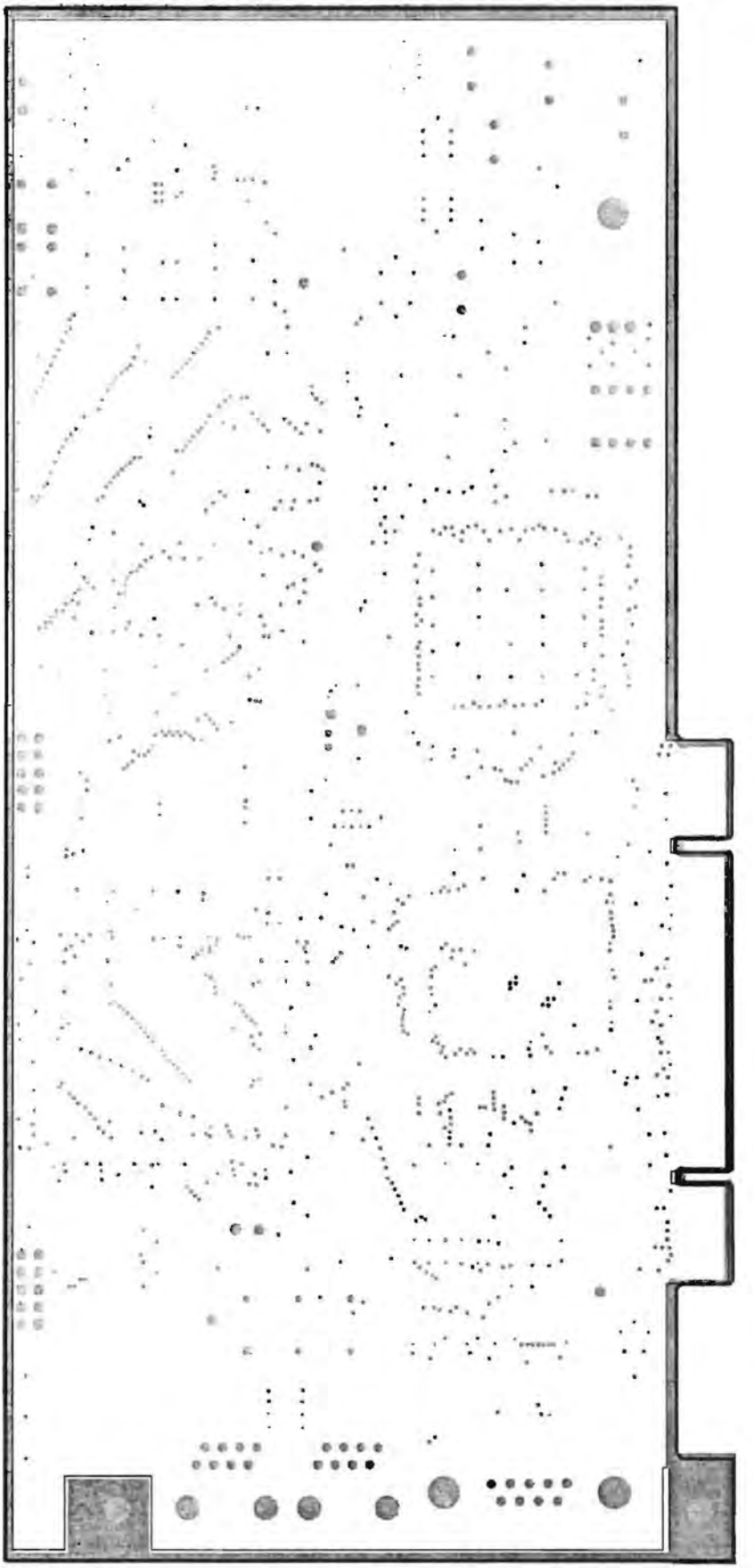
Title: Peripherals			Periph Sch
Size: A3	Number: RRSG-XXXXX		Revision: 0.1
Date: 28-Jan-2003	Sheet 8 of 9		
File: D:\Designs\Pulsar\ERIC\N01-Mods\ERIC_Mainboard.SchDoc	Drawn By:	Carl van Schaik	

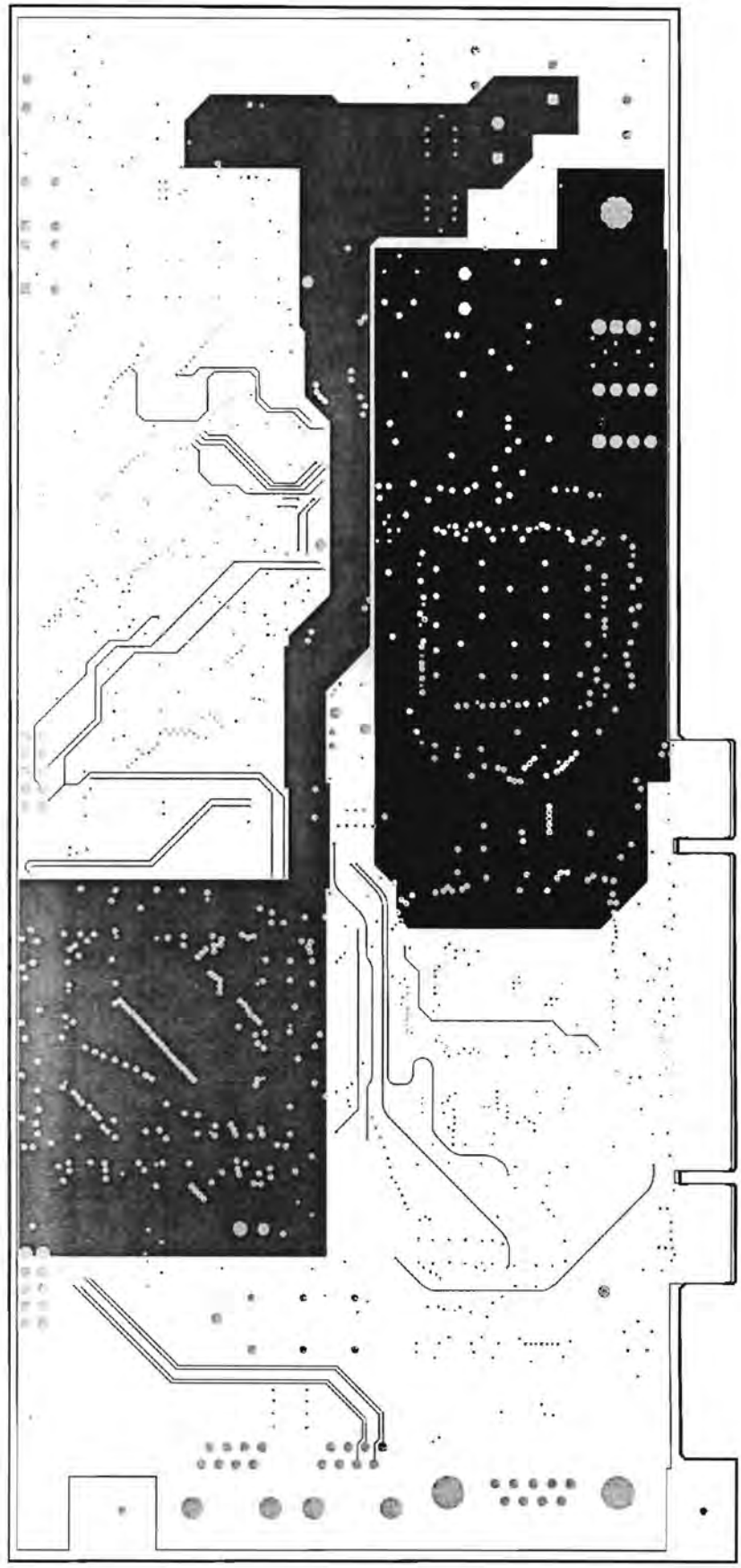


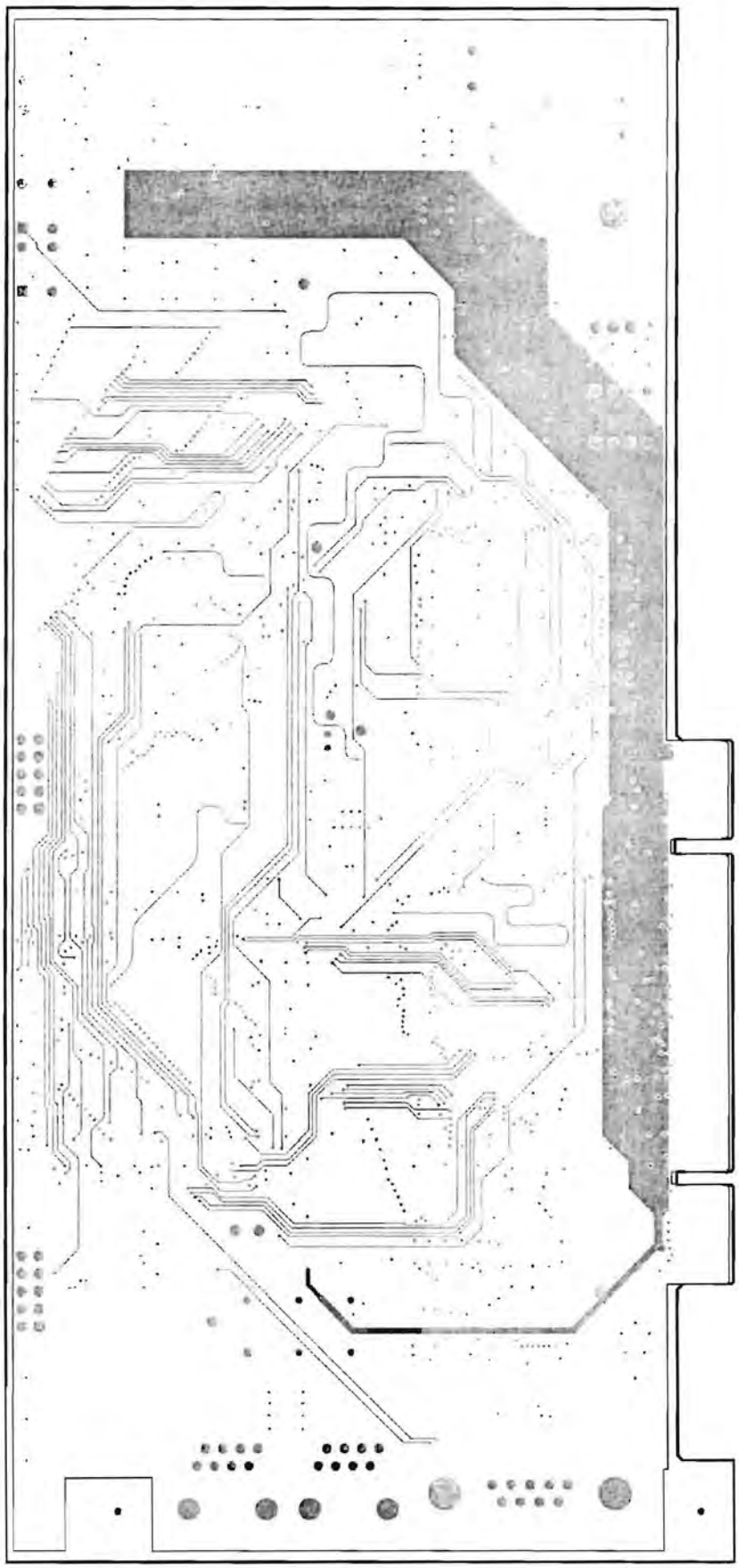
Title	FPGA Configuration Cache		ConfSRAM.Sch
Size: A3	Number: RRSG-XXXXX		Revision: 0.1
Date:	28-Jan-2003	Sheet 9 of 9	
File:	D:\Designs\Power\ERPC\N01-Mod\ERPC_Mod.SchDoc	Drawn By:	Carl van Schaik


Configuration Device
Config Sch
Size
A3
Number
RRSG-XXXXX
Revision
0.1
Sheet 1 of 10
Date:
28-Jun-2005
Drawn By:
Carl van Schaik
XC9536XL-PC44
SDIPSW
MAX811_EUS/T
LED-RED
LED-GRN
LED-YEL


PCI-MPS
System Controller
Galileo
GT-64115
SDRAM
SDRAM
XILINX
XCV800E
OpenFuel
(C) 2001
Xilinx
LT1084CT
PS-232
LVDS A
LVDS B







Appendix B

Oscilloscope Traces

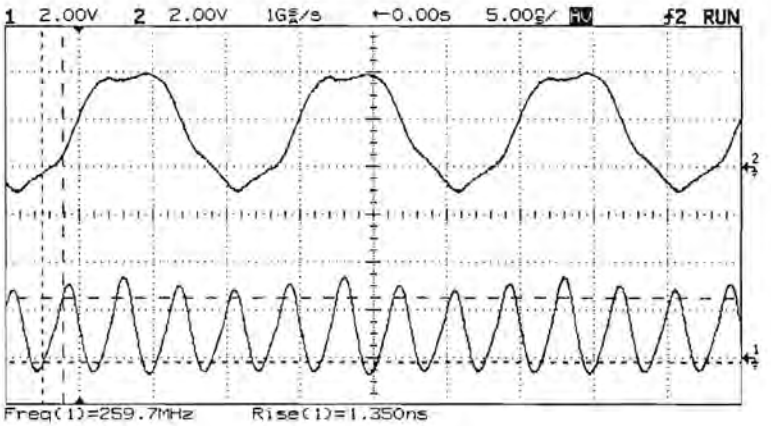


Figure B.1: 4X Clock Generation

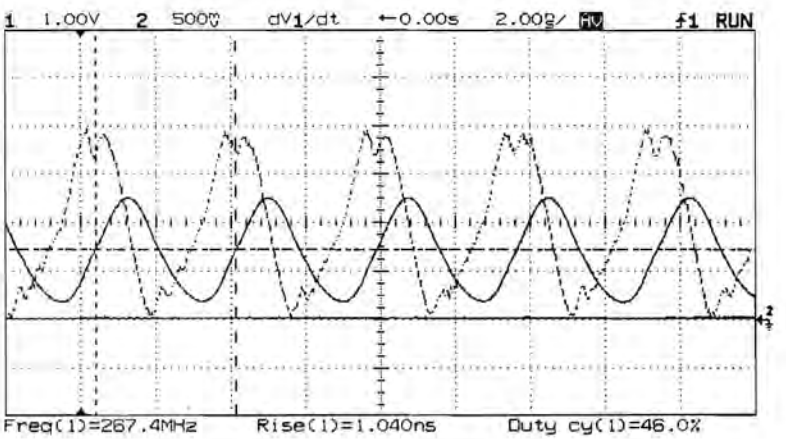


Figure B.2: 4X Clock - dV/dt (1GV/s per division)

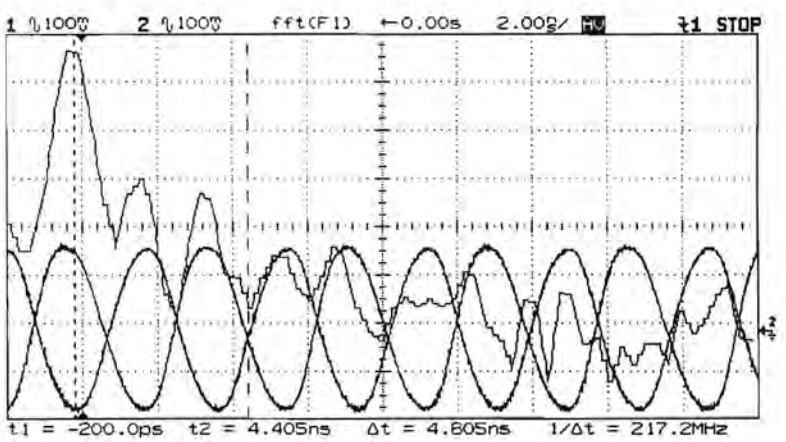


Figure B.3: LVDS Clock - FFT (266MHz)

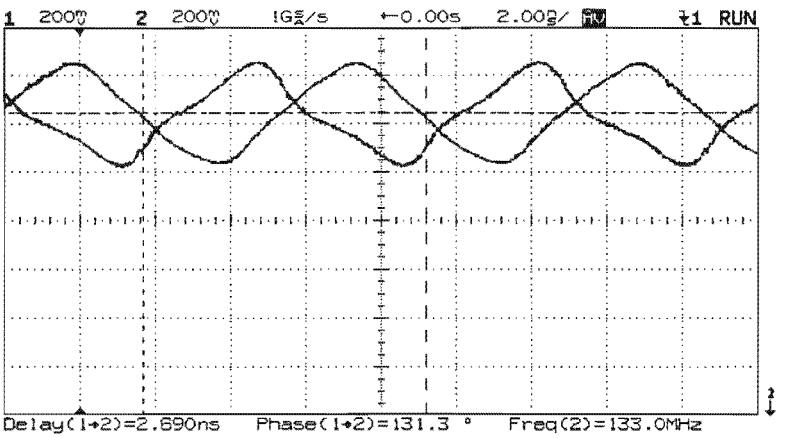


Figure B.4: LVDS Cable Delay and Signal Quality

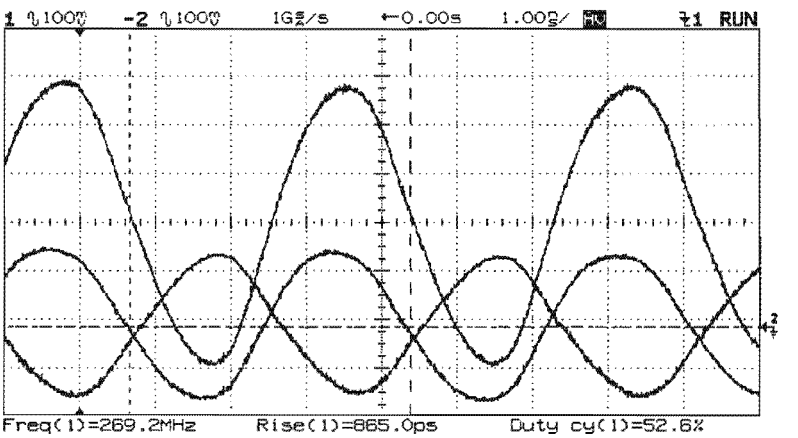
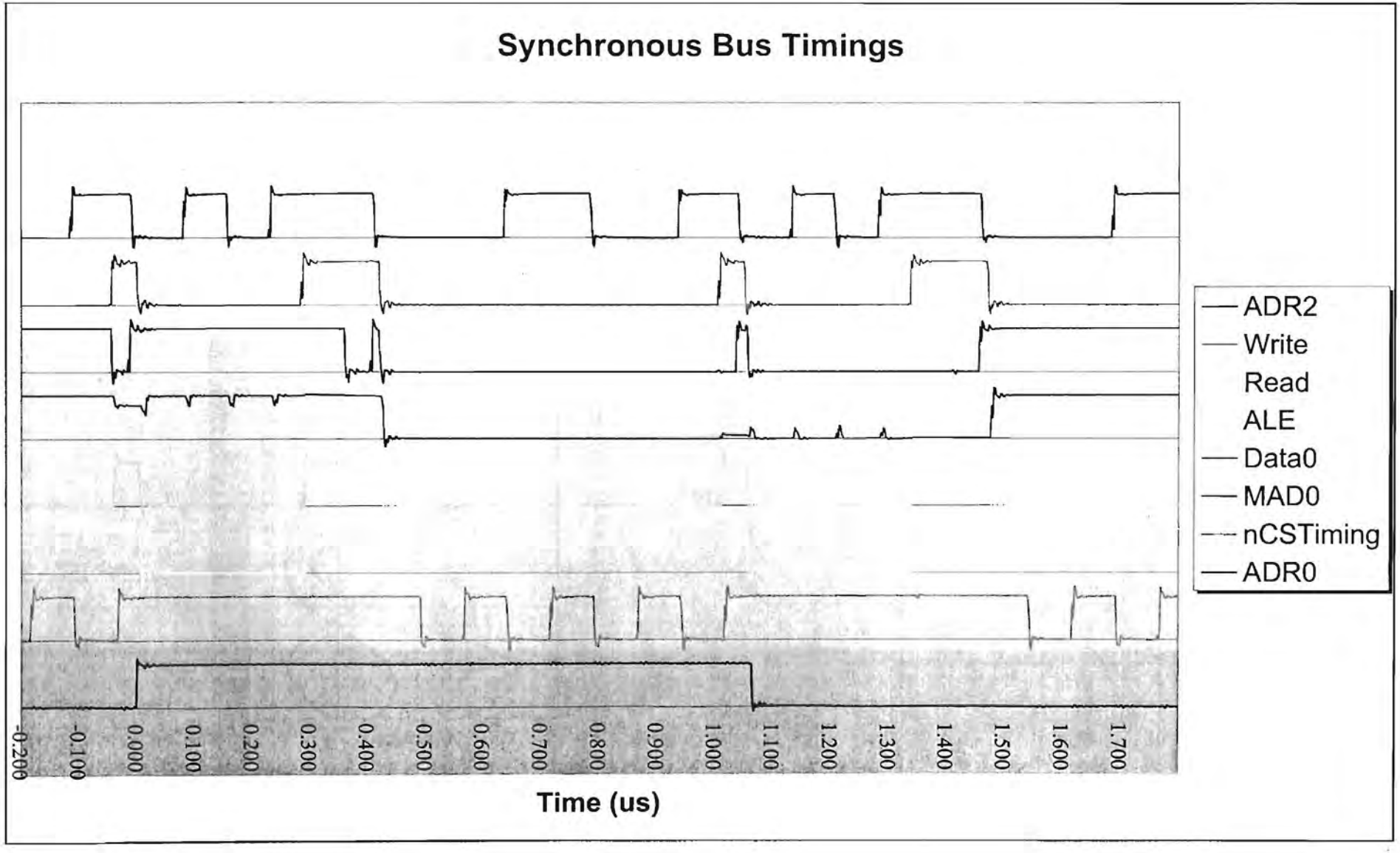


Figure B.5: LVDS Difference Voltage


Synchronous Bus Timings
ADR2
Write
Read
ALE
Data0
MAD0
nCSTiming
ADR0
-0.200
-0.100
0.000
0.100
0.200
0.300
0.400
0.500
0.600
0.700
0.800
0.900
1.000
1.100
1.200
1.300
1.400
1.500
1.600
1.700
Time (us)

Appendix C

Propane Interface

The Propane interface provides eight 1MB address windows for function implementation. Thus Address bits 20,21,22 are used for device or window addressing.

Window 0 is the Propane control interface

Windows 1..7 are designer/implementation specific.

Address 0x00 of each window is mapped to a configuration register which provides device identification and designer specifiable features.

The format for each configuration register is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
I	I	I	I	V	V	V	V	x	x	x	x	x	x	x	x
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

I - Function ID, *V* - Function Version, *x* - Design specific

Windows 0..7 correspond to internal WishBone chip-selects 0..7

The interface specification for the Propane interface on device window 0 is:

Window Address	Description	Attributes
0x00000	Configuration Register	read-only
0x00004	Interrupt Status Register (raw)	read-only
0x00008	Interrupt Mask Register	write-only
0x00100	RTC Status/Control Register	read-write
0x00104	RTC Request Address	read-write
0x00108	RTC Request Data	read-write
0x00200	Reserved - DMA Controller	
...		

Configuration Register 0x00000

Function ID: '0xE' or '1110'

Function Version: 0x1

Bits 23..8 reserved (read as '0')

Bits 7..1 Indicate function (7..1) presence in the system.

Bit 0 Function 0 presence - always '1'

Interrupt Status Register 0x00004

Bit 10 NMI Summary (Masked)

Bit 9 Interrupt Summary (Masked)

Bit 8 Interrupt Summary (Raw)

Bits 7..0 Interrupt Status of each functional unit

Interrupt Mask Register 0x00008

Bits 23..16 Logic 'AND' mask to generate NMI.

Bits 7..0 Logic 'AND' mask to generate Interrupt

RTC Status/Control Register 0x00100

Bit 9 Alarm 2 Status

Bit 8 Alarm 1 Status

Bit 7 Alarm 2 Enable

Bit 6 Alarm 1 Enable

Bit 5 Enable alarm polling

Bit 4 Send 'RTC Write' request (write-only)

Bit 3 Send 'RTC Read' request (write-only)

Bit 2 I^2C Data valid

Bit 1 $I^2C\ Interface$ busy

Bit 0 I^2C User request busy

Various other Propane functional unit interfaces were specified:
Propane UART Interface:

Window Address	Description	Attributes
0x00000	Configuration Register	read-write
0x00004	Transmit / Receive Register	read(receive)-write(transmit)
0x00008	Status Register	read-write

Configuration Register 0x00000

Function ID: '0xA' or '1010'

Function Version: 0x1

Bit 21 UART Enable

Bits 19..8 Baud rate (bits 11..0)

Bit 7 Receive interrupt enable

Bit 6 Transmitter empty interrupt enable

Bits 5..4 RX FIFO interrupt trigger level

Bit 3 Protocol bits - '0' - 8-bits, '1' - 7-bits

Bit 2 Stop bits - '0' - one stop bit, '1' - two stop bits

Bit 1 Even parity

Bit 0 Parity enable

Transmit / Receive Register 0x00004

Bits 7..0 Data bits

Status Register 0x00008

Bit 11 Frame Error

Bit 10 Parity Error

Bit 9 Receiver empty

Bit 8 Transmitter empty

Bits 7..4 Receive FIFO contents count

Bits 3..0 Transmit FIFO contents count

LED Display Interface:

Window Address	Description	Attributes
0x00000	Configuration Register	read-write

Configuration Register 0x00000

Function ID: '0xB' or '1011'

Function Version: 0x1

Bits 23..16 LED 7..0 Available status

Bits 7..0 LED 7..0 Output register

Propane Sigma-Delta DSP Interface:

Window Address	Description	Attributes
0x00000	Configuration Register	read-write
0x00004	FIFO Input Register	write-only
0x00008	DSP Rate Register	read-write

Configuration Register 0x00000

Function ID: '0x5' or '0101'

Function Version: 0x1

Bit 17 FIFO Full

Bit 16 FIFO Empty

Bits 15..0 FIFO contents count

Propane DSP Interface:

Window Address	Description	Attributes
0x00000	Configuration Register	read-write

Configuration Register 0x00000

Function ID: '0xF' or '1111'

Function Version: 0x0 - (DCT)

Appendix D

Source Code and Datasheets

Due to the size of the associated source code and datasheets, this information has been omitted from this report.
Copies of the full source code, development tool and datasheets are provided on CDROM format. Available on request.